VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro neuronovou síť Flexible Neural Tree

Flexible Neural Tree Framework

2013

Pavel Piskoř

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Pavel Piskoř**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Framework pro neuronovou síť Flexible Neural Tree
Flexible Neural Tree Framework

Zásady pro vypracování:

Cílem práce je implementovat knihovnu pro práci s neuronovou sítí Flexible Neural Tree (FNT). Knihovna bude založena na frameworku .NET.
1. Proveďte rešerši současného stavu poznání.
2. Objektově orientovaný návrh knihovny pro FNT.
3. Implementace této knihovny.
4. Testy výkonnosti implementované neuronové sítě na ukázkových příkladech.

Seznam doporučené odborné literatury:

Yuehui Chen, Ajith Abraham. Tree-Structure based Hybrid Computational Intelligence: Theoretical Foundations and Applications (Intelligent Systems Reference Library). Springer 2009. ISBN 978-3642047381.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Doc. Mgr. Jiří Dvorský, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 31. července 2013 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 31. července 2013 .....

Na tomto místě bych rád poděkoval panu Doc. Mgr. Jiřímu Dvorskému, Ph.D. za poskytnuté rady při vypracovávání této práce a za čas, který mi věnoval při odborných konzultacích.

Abstrakt

Práce se zaměřuje na neuronovou síť zvanou Flexibilní neuronový strom. Popisuje nejen strukturu a parametry sítě samotné, ale také postupy, jak nalézt optimální neuronovou síť pro konkrétní úlohu. Výsledkem práce je implementace knihovny v prostředí .NET, která umožňuje automaticky nalézt vhodnou neuronovou síť pro konkrétní zadání.

Klíčová slova: Flexibilní neuronový strom, neuronová síť, evoluční algoritmy, genetické programování, metoda zpětného šíření

Abstract

The work is focused on a neural network which is called the Flexible Neural Tree. It describes not only network structure and parameters, but it also describes the techniques, how to find out the optimal neural network for a specific task.

Keywords: Flexible Neural Tree, neural network, evolutionary algorithms, genetic programming, back propagation

Seznam použitých zkratek a symbolů

FNT	–	Flexible Neural Tree
PE-FNT	–	Parallel evolving algorithm for FNT
FBBFNT	–	Flexible Beta Basis Function Neural Tree
GP	–	Genetic Programming
MEP	–	Multi expression programming
EIP	–	Extended Immune Programming
PIPE	–	Probabilistic Incremental Program Evolution
DE	–	Differential Evolution
PSO	–	Particle Swarm Optimization
HBFOA	–	Hybrid Bacterial Foraging Optimization Algorithm
MSE	–	Mean Square Error
RMSE	–	Root Mean Square Error
NMSE	–	Normalized Mean Square Error

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Výzkum lidského mozku zaujal vědce natolik, že se rozhodli vytvořit jeho umělý model. Jelikož je mozek velice složitý systém a jeho funkce nebyly dosud plně prozkoumány a popsány, bylo a je téměř nemožné plně napodobit fungování mozku. Jeho základní části je rozsáhlá síť mnoha navzájem propojených neuronů. Zprvu tedy byly snahy o vytvoření matematického modelu neuronu a jednoduché sítě, skládající se z několika takovýchto neuronů. Na počátku čtyřicátých let dvacátého století představili *McCuloch* a *Pitts* první model umělého neuronu a jednoduchou síť, skládající se z těchto neuronů. Tato síť byla nemodifikovatelná. To znamená, že při vytváření se zvolila pevná struktura sítě a nastavily se jí na pevno hodnoty parametrů. Tito pánové následně prohlásili, že při vhodně zvolené struktuře sítě a vhodném nastavení hodnot parametrů dokáže taková síť provádět stejné úlohy jako klasický počítač. Objevuje se tedy další důvod k výzkumu umělých neuronových sítí. Tedy nejen poznávání, jak funguje lidský mozek, ale i inspirace nového způsobu řešení výpočetních problémů. Potíž však byla v nalezení vhodných hodnot parametrů tohoto modelu pro řešení konkrétních úloh. Později byly objeveny algoritmy, které dokáží najít tyto hodnoty automaticky. Nazývají se učící nebo trénovací algoritmy. Tímto byl vyřešen problém nastavování hodnot parametrů a zároveň se otevřely další možností v této oblasti. Objev učících algoritmů tak zvedl zájem dalších vědců a nadšenců v oblasti výzkumu neuronových sítí.

Protože jeden typ sítě není vhodný pro řešení všech typů úloh, vznikaly další a další typy neuronových sítí, které se lišily topologií neboli vnitřní strukturou. Neuronové sítě tak byly schopny řešit problémy jako klasifikace, predikce, aproximace, atd. Bylo tedy vytvořeno několik typů neuronových sítí a pro ně nalezeny vhodné trénovací algoritmy učící neuronové sítě řešit konkrétní problém. Stále se však musela manuálně navrhovat vhodná struktura sítě. Otázky návrhu vhodné struktury neuronové sítě se týkají například počtu použitých neuronů v síti. Bude-li počet neuronů příliš malý, nebude síť schopna správného zobecňováni. Nebude totiž schopna naučit se všem trénovacím vzorům. Na druhou stranu, pokud by byl počet neuronů příliš velký, opět nebude síť schopna správného zobecňováni. Tentokrát ale z jiného důvodu. Síť se naučí všem trénovacím vzorům, ale s velkou přesností. Bude tedy umět přesně odpovídat jen na předem naučené vzory. Další otázkou návrhu vhodné struktury je topologie neboli vzájemné propojení těchto neuronů. I pro tuto problematiku se našlo několik způsobů řešení, ale algoritmy jsou složité a ne vždy vedou k optimálnímu návrhu struktury sítě. Tuto otázku částečně řeší jednotlivé typy neuronových sítí, které kladou jistá omezení na propojení neuronů. Každý typ sítě se tedy hodí pro jinou třídu problémů.

Předmětem této práce je implementace neuronové sítě typu Flexibilní neuronový strom. Algoritmy použité v implementaci řeší oba výše zmiňované problémy návrhu automaticky. Na základě předložení konkrétního výpočetního problému naleznou vhodnou strukturu flexibilního neuronového stromu řešícího tento problém a nastaví všechny potřebné parametry na správné hodnoty. Další části práce se věnují základnímu přehledu

typů neuronových sítí, současnému stavu poznání ohledně flexibilních neuronových stromů a také testovacím příkladům, ukazujícím možnosti použití tohoto typu neuronové sítě.

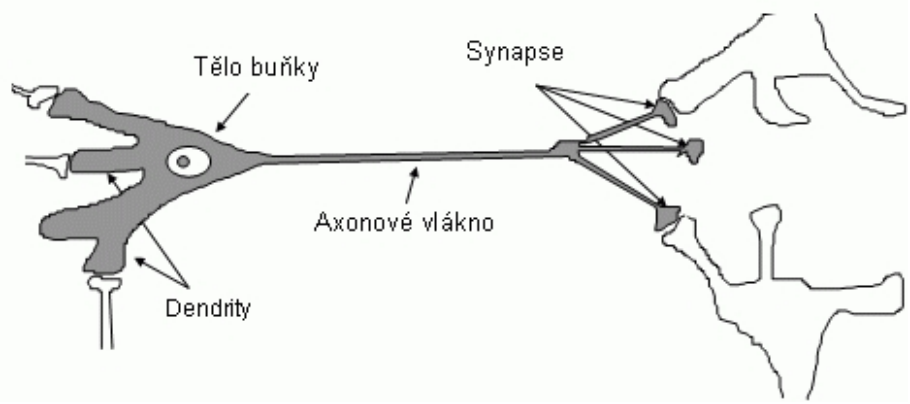
2 Neuronové sítě

Pod pojmem neuronová síť ve výpočetní technice rozumíme umělou neuronovou síť, která se snaží napodobit biologickou neuronovou síť. Myšlenka použití neuronové sítě k řešení složitých úkolů se inspiruje přírodou. V přírodě lze nalézt spoustu živočichů, disponujících neuronovou sítí, kteří musí neustále řešit nějaké úkoly. Tuto biologickou neuronovou síť nazýváme mozek. Bylo vypozorováno, že i živočichové s relativně malou neuronovou sítí, dokáží řešit složité situace (úkoly). I když se doposud umělé neuronové sítě zdaleka nevyrovnají těm biologickým, jsou jejich výsledky použitelné pro řešení různých výpočetních problémů. Podkladem pro studium využití neuronových síti se tedy stala činnost mozku. Ačkoli mají neuronové sítě vysoký výpočetní výkon v reálném čase, nedají se použít na řešení všech výpočetních problémů, protože dávají nepřesné výsledky. Jejich výhoda spočívá ve schopnosti naučit se řešit daný problém na základě předkládání takzvaných trénovacích vzorů. To má tu výhodu, že nemusíme znát přesný algoritmus řešení. My jen „ukážeme“ neuronové síti několik příkladů a síť si sama najde algoritmus řešení, pokud nějaký existuje. Neuronové sítě jsou tedy s výhodou používány všude tam, kde neznáme algoritmus řešení nebo je algoritmus příliš složitý pro matematické formulování problému. Neuronové sítě tak najdou uplatnění například v oblastech optimalizace topologie, systémech pro rozhodování, komprese a kódování, predikce časových řad a také v oblasti umělé inteligence.

Pro lepší pochopení si můžeme neuronovou síť zjednodušeně představit jako černou skříňku. Ve fázi učení předkládáme na jednom konci skříňky příklady a na druhém konci jejich řešení. Skříňka na to reaguje změnou svého vnitřního stavu. Tuto činnost opakujeme dokola po určitý čas. Po několika takových cyklech je skříňka schopna naučit se odpovídat na naše příklady sama. Neuronová síť tedy disponuje schopností učit se. Po ukončení fáze učení je skříňka schopna odpovídat nejen na naučené příklady, ale i na příklady, které jí v průběhu učení předloženy nebyly. Na nenaučené příklady však skříňka odpovída s menší přesností než na ty, které ji byly předkládány ve fázi učení, neboť je „odhaduje“ na základě toho, co se naučila. Druhou schopností neuronové sítě je tedy zobecňování neboli generalizace. Síť je schopná určité abstrakce. Tyto dvě schopnosti sítě jdou však proti sobě. Síť, která se naučí odpovídat na předkládané příklady s velkou přesností, má sníženou schopnost zobecňování. Někdy se tento stav nazývá také „přeučení sítě“. Naopak síť, která umí dobře zobecňovat, tedy odpovídat i na nepředložené příklady, bude odpovídat s menší přesností.

2.1 Biologický neuron

Základem každé neuronové sítě jsou neurony. Neurony jsou mezi sebou propojeny a slouží k přenosu, zpracování a uchování informací. Vzorem pro mamtematický model neuronu je biologický neuron. Ten se skládá z těla zvaného *soma*, ze kterého vybíhá několik tisíc výběžků nazývaných *dendrity* a vlákno zvané *axon*. Dendrity jsou dlouhé pouze několik milimetrů a tvoří vstupy neuronu. Axon nabývá délky až okolo 60 centimetrů a představuje výstup neuronu. Konec axonu je rozvětven do tzv. synapsí, které tvoří sty-



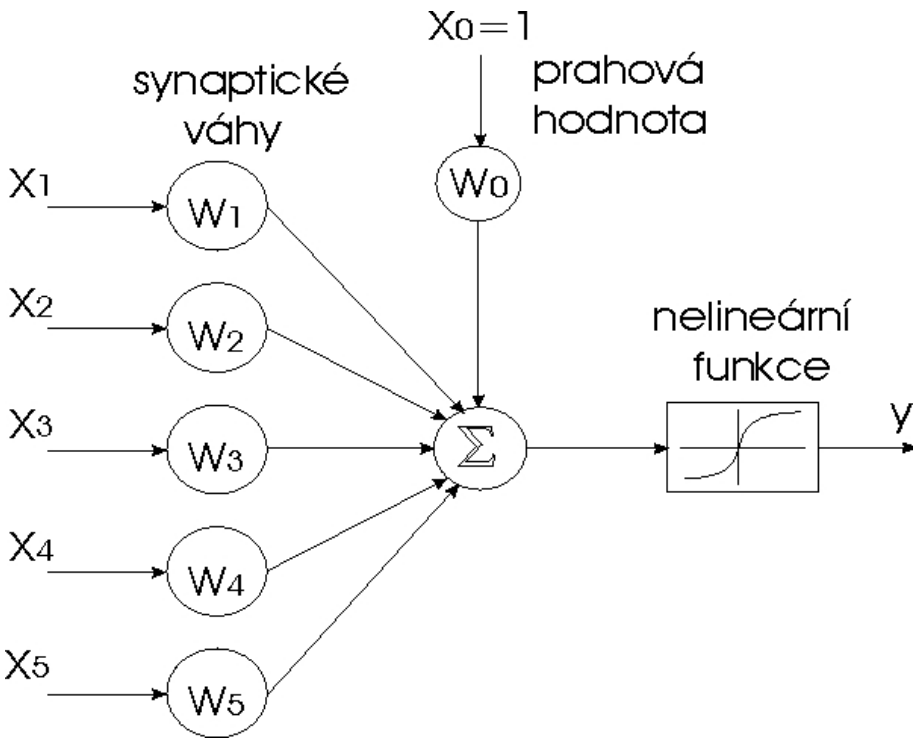
Obrázek 1: Model biologického neuronu - převzato z [13]

kové plošky. Synapse pak přiléhají na dendrity dalších neuronů a tím utvářejí spojení mezi neurony. Synapse jsou dvojího druhu. *Excitační*, které zesilují výstupní signál a *inhibiční*, které tlumí výstupní signál.

Biologický neuron má za úkol shromáždit informace ze svých vstupů, zpracovat je a poslat na výstup. Jelikož výstup (axonové vlákno) je na svém konci rozvětven do mnoha synapsí, může mít neuron obecně mnoho výstupů. Zpracování vstupních informací spočívá v jejich sumaci (sečtení, případně odečtení) a následném porovnání s hodnotou prahu. Pokud výsledná hodnota sumace vstupů zvaná *potenciál* překročí prahovoou úroveň, je neuron aktivován, čili vyšle signál na výstup. Jelikož se tento děj (sumace, prahování a aktivace) s určitou frekvenci opakuje, přenáší se informace posloupností impulsů, což nazýváme frekvenční modulací. Protože synapse neuronu mohou výstupní signál zesilovat nebo utlumovat, obdrží následující neurony, s tímto neuronem spojené, různé velikosti výstupního signálu na svých vstupech. Vlastnost synapse zesílení nebo utlumení signálu je vyjádřena jako synaptická váha. Váhy jednotlivých synapsí se s časem mění. Má-li konkrétní spoj z jednoho neuronu do druhého větší význam než spoje do jiných neuronů, posílí se váha jeho synapse. Naopak, má-li spoj menší význam, projeví se to snížením synaptické váhy. Změny synaptických vah na spojích mezi jednotlivými neurony jsou výsledkem procesu učení neuronové sítě.

2.2 Formální neuron

Umělý neboli syntetický neuron je zjednodušený matematický model biologického neuronu, nazývaný také formální neuron. Matematickou formulaci neuronu zavádíme proto, abychom mohli neuron simulovat pomocí počítače a používat jej pro řešení úloh. Struktura formálního neuronu je zobrazena na obrázku č. 2. Formální neuron (dále jen neuron) obsahuje n vstupů $x_1, \ldots, x_n$, které představují dendrity. Vstupy jsou ohodnoceny váhami $w_1, \ldots, w_n$, které jsou analogií synaptických vah. Vážená suma vstupních hodnot pak představuje vnitřní potenciál neuronu. Odečtením prahu od tohoto potenciálu zís-



Obrázek 2: Model formálního neuronu - převzato z www.elektrorevue.cz

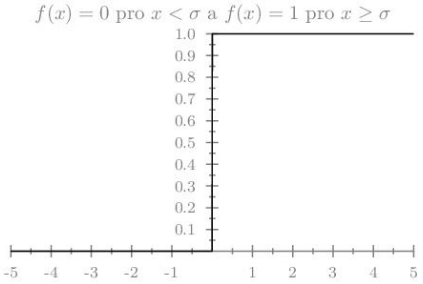
káme hodnotu, která je dále zpracována aktivační funkcí neuronu. Výsledkem aktivační funkce je výstupní hodnota neuronu. Hodnoty všech vstupů, vah, prahů a výstupů bývají nejčastěji reálná čísla.

Matematicky to můžeme zapsat takto:

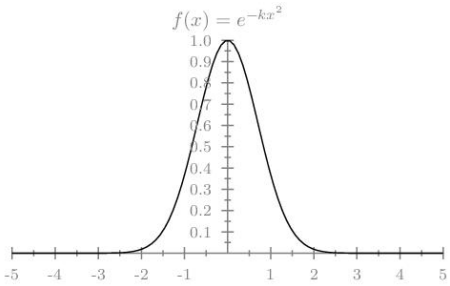
$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i - w_0\right) \tag{1}$$

Kde:
x_i - jsou hodnoty vstupů
w_i - jsou hodnoty vah
w_0 - je hodnota prahu
n - je počet všech vstupů
σ - je aktivační funkce
y - je výstup

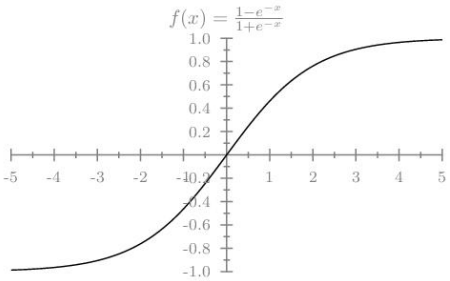
Matematických funkcí, které lze použít jako aktivační funkce neuronu, existuje více. Konkrétní použitá funkce pak dává neuronu, a tím i celé neuronové síti, konkrétní chování. Některé funkce používané v modelech neuronů jsou vyobrazeny na obrázcích 3 - 6.



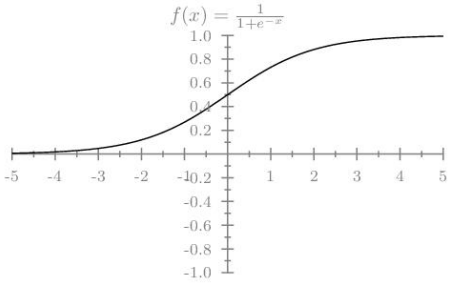
Obrázek 3: Průběh funkce Ostrá nelinearita



Obrázek 4: Průběh funkce Radiální báze



Obrázek 5: Průběh funkce Hyperbolický tangens



Obrázek 6: Průběh funkce Standardní sigmoida

2.3 McCuloch - Pittsův model neuronu

První model formálního neuronu byl roku 1943 představen *McCulochem* a *Pittsem*. Byl to *binární prahový neuron*, který používal na svých vstupech i na svém výstupu pouze hodnoty „0“ a „1“. Hodnota „1“ na vstupu neuronu označovala, že přichází vzruch, hodnota „0“ pak, že vzruch nepřichází. Excitační synapse byla vyjádřena váhovou hodnotou „+1“, kterou se vynásobil vstup a inhibiční synapse hodnotou „-1“. Tyto váhové hodnoty byly pevně dané. Jako aktivační funkce byla použita skoková funkce zvaná *Ostrá nelinearita*, obr. 3. Hodnota „1“ se tedy na výstupu objevila pouze pokud vážený součet všech vstupů přesáhl prahovou hodnotu. Tento model se od skutečného biologického neuronu sice podstatně liší, ale i přesto je schopen řešit některé typy úloh.

„McCulloch a Pitts dokázali, že synchronní pole takovýchto neuronů je v principu schopno libovolného výpočtu a tudíž může provádět stejné výpočty jako digitální počítač.“[7].

Problém však nastává s volbou vhodné struktury sítě a s nastavením prahů a vah jednotlivých neuronů. Sítě z těchto neuronů se totiž konstruovaly jako neměnné. Jinými slovy, nebyly schopné učení. Také vlastnosti tohoto typu neuronu, jako např. skoková

změna výstupu a jedna úroveň výstupního signálu oproti spojité změně výstupu a sledů impulsů na výstupu u biologického neuronu, nejsou vhodné pro některé typy úloh. Z těchto důvodů se začal hledat jiný model neuronu.

2.4 Perceptron

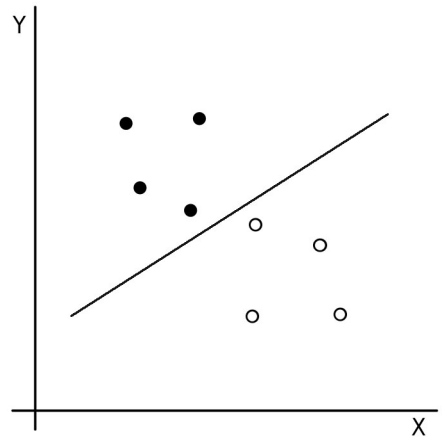
V roce 1958 tedy přišel na scénu nový model neuronu nazvaný *Perceptron*, jehož autorem je Frank Rosenblatt. Jen pro uřesnění, slovo Perceptron se používá pro název určitého typu neuronu a taky pro neuronovou síť složenou z těchto neuronů. Rosenblatt přišel na jednoduchý učící algoritmus, kterým dovedl učit jednovrstvou perceptronovou síť. Základem toho byla změna v konstrukci sítě, že synaptické váhy již nebyly pevně dané, ale mohly se měnit podle potřeby. Tato změna přidala schopnost síti učit se a tímto se model přiblížil svému biologickému vzoru. Princip spočíval v tom, že nejdříve zaznamenal odpovědi (výstupní hodnoty) všech neuronů na konkrétní podnět a poté měnil váhy spojení takto: Pokud byla odpověď neuronu správná, váhy neměnil. Pokud byla odpověď špatná, měnil váhy všech spojů, které do něj vedly. Měl-li být neuron aktivní, posílil váhy. Měl-li být neaktivní, zeslabil je. Jednovrstvá perceptronová síť však dokáže řešit jen některé jednoduché typy úloh a proto se jí nedostávalo příliš zájmu. Nedokáže například řešit ani tak základní úkol, kterým je operace XOR. Obecně je to tak, že nedokáže řešit lineárně neseparovatelné problémy, jak ukazuje obrázek 8 v kapitole 2.5.

Teprve po objevení algoritmu schopného učit i vícevrstvou síť, se vývoj neuronových sítí podstatně změnil. Tento algoritmus se nazývá *back-propagation*, což v překladu znamená metoda zpětného šíření a představili jej v roce 1986 David E. Rumelhart, Geoffrey E. Hinton a Ronald J. Williams. Aby mohli algoritmus použít, provedli několik změn. Vstupy, výstupy a váhy už nepracovaly jen s binárními hodnotami, ale používaly reálná čísla. Přestali používat skokovou aktivační funkci a nahradili ji spojitou. Nejčastěji používaná funkce u perceptronu je standardní sigmoida, obr. 6. Struktura sítě dostala omezující pravidla jako například: Síť se skládá z několika vrstev. První vrstvu tvoří vstupní a poslední vrstvu výstupní neurony. Mezi těmito vrstvami mohou být další (skryté) vrstvy. Všechny neurony jedné vrstvy jsou jednosměrně propojeny se všemi neurony následující (vyšší) vrstvy. Neurony jedné vrstvy nejsou mezi sebou spojeny a neexistují ani spojení přes několik vrstev. Síť je dopředná, což znamená že signály se postupně šíří pouze jedním směrem od vstupní vrstvy k vrstvě výstupní.

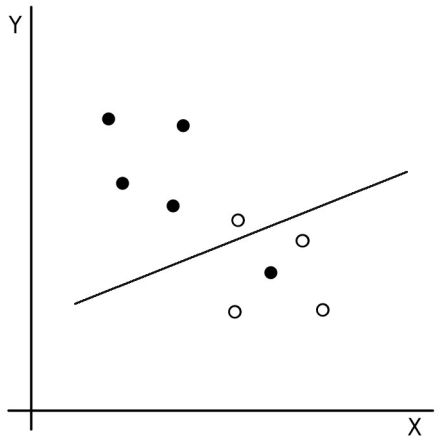
2.5 Geometrická interpretace perceptronu

Jeden neuron sám o sobě může řešit pouze ty problémy, které jsou lineárně separabilní. Tuto vlastnost neuronu nám názorně ukazuje jeho geometrická interpretace na obrázcích 7 a 8. Vstupy neuronu představují souřadnice bodu v n-rozměrném prostoru. Zde je pro jednoduchost použit dvou rozměrný prostor, čili rovina. V tomto případě má rovnice 1 následující tvar:

$$y = \sigma(w_1 x_1 + w_2 y_2 - w_0) \tag{2}$$



Obrázek 7: Lineární separabilita neuronu - množiny bodů, které jdou oddělit jedinou přímkou



Obrázek 8: Lineární separabilita neuronu - množiny bodů, které nejdou oddělit jedinou přímkou

Rovnici $w_1x_1 + w_2y_2 - w_0 = 0$ lze matematickými úpravami a substitucí proměnných zjednodušit na tvar $y = ax + b$, což vlastně znamená, že se na ni můžeme dívat jako na rovnici přímky. Přímka nám rozdělí rovinu na dvě poloroviny. V obecném případě pro n-rozměrný prostor mluvíme o rovnici nadroviny, která dělí prostor na dva poloprostory. Máme-li jako aktivační funkci neuronu použitou ostrou nelinearitu, pak pro body na jedné straně přímky nastaví neuron svůj výstup na hodnotu nula a pro body na druhé straně přímky včetně bodů ležící na přímce nastaví výstup na hodnotu jedna. Bodem rozumíme určitou kombinaci vstupních hodnot neuronu, která jak bylo řečeno výše, představuje souřadnice bodu.

Co tedy neuron dělá, je dělení prostoru na dva poloprostory. Jinými slovy, umí oddělit dvě množiny bodů v prostoru. Procesem učení, při kterém se mění hodnoty jednotlivých parametrů neuronu, tedy říkáme do kterého poloprostoru konkrétní body patří. V případě dvou rozměrného prostoru (neuronu se dvěma vstupy), který je na obrázku 7, se procesem učení mění hodnoty vah w_1, w_2 a hodnota prahu w_0. Hodnoty vah vlastně určují úhel dělící přímky, který svíra s osou x neboli její natočení a hodnota prahu určuje posun přímky na ose y.

Jak je napsáno výše, jeden neuron může řešit pouze problémy, které jsou separabilní. To znamená, že prostor se dá rozdělit na dva poloprostory jednou přímkou tak, aby každá množina bodů ležela v jiném poloprostoru. Pokud se množiny bodů nedají rozdělit do dvou poloprostorů jedinou přímkou, nelze tyto problémy řešit jedním neuronem. Tento stav zachycuje obrázek 8.

Použijeme-li neuronů více, získáme tak více dělících nadrovin. Ve dvourozměrném pro-

storu to znamená, že získáme více přímek, které nám budou dělit tento prostor. Máme-li více přímek, můžeme z jejích částí poskládat křivku. Křivka pak dokáže rozdělit i složitější obrazec bodů v rovině. Můžeme tak v rovině „vykreslit“ třeba trojúhelník nebo čtverec, kterým oddělíme určitou množinu bodů od jiné množiny. To, jak složitý problém dokáže konkrétní síť řešit, závisí na její struktuře. Samozřejmě schopnosti neuronových sítí nejsou omezeny jen na problémy přímo související s dělením prostoru. Když změníme význam vstupních a výstupních hodnot, principiálně sice bude síť pořád provádět dělení prostoru, ale řeší nám problém, který je na první pohled zcela jiný. Například jeden vstup může představovat teplotu vody v nádobě a druhý vstup tlak okolního vzduchu. Výstup pak může udávat, jestli při určité kombinaci teploty vody a tlaku okolního vzduchu bude voda v nádobě ve varu nebo ne.

2.6 Učící algoritmus zvaný Back-propagation

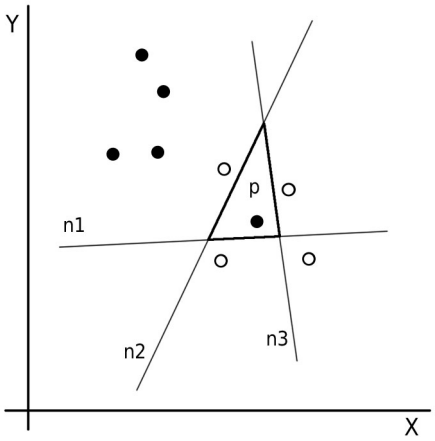
Jak už bylo zmíněno výše, neuronové sítě mají schopnost učit se. Rozlišujeme dva typy učení neuronových síti. Učení s učitelem a učení bez učitele. Učení pomoci algoritmu back-propagation patří do skupiny učení s učitelem, protože využívá zpětnou vazbu při učení na jejímž základě mění hodnoty synaptických vah. Učení bez učitele bude vysvětleno později v kapitole 2.7.2 Kohonenovy mapy. Před učením samotným si musíme připravit tzv. trénovací množinu. Trénovací množina je sada pravidel, které chceme neuronovou síť naučit. Takovýmto pravidlům říkáme trénovací vzory. Každý trénovaci vzor se skládá ze dvou částí. První část obsahuje „podnět“ neboli hodnoty, které předložíme na vstup sítě, druhá část obsahuje očekávanou „odezvu“ na tento podnět, tj. očekávané hodnoty na výstupu sítě. Základní princip algoritmu učení back-propagation tedy spočívá v postupném předkládání podnětů trénovacích vzorů na vstupy sítě a vyhodnocování odezvy sítě. Na základě odezvy sítě jsme schopni určit chybu, které se síť dopustila, což je vlastně rozdíl mezi skutečnými hodnotami na výstupu a očekávanými hodnotami. Podle chyby sítě pak měníme hodnoty vah v určitých poměrech. Tento postup opakujeme neustále dokola, až se síť trénovací vzory naučí nebo do vyčerpání předem stanoveného počtu učících cyklů.

V prvním kroku při učení sítě tedy předložíme jeden podnět trénovacího vzoru na vstup sítě a necháme jej „prošířit“ na výstup, kde obdržíme odezvu. Tomuto kroku se říká dopředné šíření a jeho výpočet probíhá podle již zmíněné rovnice 1. Ve druhém kroku vypočteme chybu výstupu. Tu můžeme vypočítat několika způsoby. Buď jako celkovou chybu sítě přes všechny výstupní neurony nebo třeba pro každý neuron zvlášť. Duležité je, že ve výsledku máme připravenou velikost chyby pro každý výstupní neuron. Ve třetím kroku provedeme zpětné šíření chyby z výstupu na vstupy, při kterém si buďto zapamatujeme vypočtené změny pro synaptické váhy nebo rovnou váhy měníme. Při zpětném šíření postupujeme tak, že vypočtené chyby předložíme na výstupní neurony a prošíříme je zpět ke vstupům. Pro výpočty nepoužíváme aktivační funkci samotnou ale její derivaci. Nyní známe pro každý neuron hodnotu chyby, které se dopustil, a tu potřebujeme v určitém poměru předat na jeho vstupy a pak dál neuronům do nižší vrstvy. K tomuto účelu použijeme právě derivaci aktivační funkce a gradietní metodu nazývanou „sestup svahem“. Tímto tedy vypočteme pro každý vstup neuronu konkrétní poměr

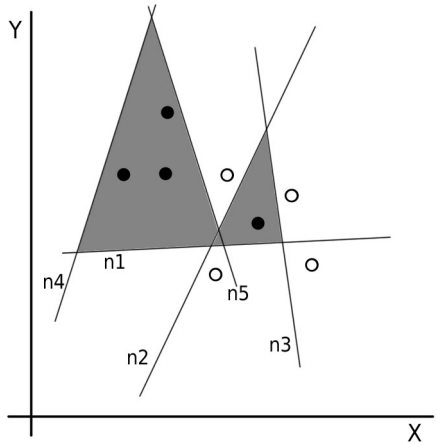
chyby, kterým tento vstup přispěl k celkové chybě a předáme jej neuronu v nižší vrstvě. Každý neuron v nižší vrstvě sečte všechny části chyby obdržené od neuronů z vyšší vrstvy a provede stejný postup výpočtu aby předal částečné chyby dál do nižších vrstev. Přesný popis algoritmu back-propagation včetně všech potřebných vzorců a obrázků pro výpočet lze najít v publikacích [13] , [12] a [2].

2.7 Typy neuronových sítí

2.7.1 Perceptronová síť



Obrázek 9: Logická funkce AND realizována neurony 3. vrstvy sítě typu Perceptron



Obrázek 10: Logická funkce OR realizována neurony 4. vrstvy sítě typu Perceptron

Základem pro perceptronovou síť je model neuronu zvaný Perceptron, jak již bylo popsáno výše. Na strukturu sítě je kladeno několik pravidel, které také byly popsány dříve. Z nich zopakujeme jen jedno. Síť se skládá z několika vrstev. Minimálně tedy ze dvou, z nichž jedna je vstupní a druhá výstupní. První vrstva neuronů (vstupní) pouze předává hodnoty vstupů na svůj výstup. Druhá vrstva svými neurony vytváří dělící nadroviny a výstup každého neuronu druhé vrstvy tak vyjadřuje, do kterého poloprostoru patří vstupní vektor (kombinace vstupních hodnot). Neurony třetí vrstvy mohou být nastaveny tak, že představují průnik některých poloprostorů z druhé vrstvy. Jinými slovy, každý neuron třetí vrstvy může symbolizovat jednu množinu bodů v prostoru, které spolu nějak souvisí. Tímto vlastně neurony třetí vrstvy realizují logickou funkci „AND“. Tato funkce je zobrazena na obrázku 9. Rovina obsahuje 3 přímky n1, n2 a n3, které představují dělící nadroviny tří různých neuronů. Průnik těchto přímek označený písmenem „p“ tvoří konvexní oblast (trojúhelník), která obsahuje jeden bod. Tento bod reprezentuje jeden vstupní vektor. Máme-li množinu bodů spolu souvisejících rozprostřenou v prostoru tak, že nejde uzavřit jednou konvexní oblasti, musíme použít více těchto oblastí

pomoci dalších neuronů. Neurony čtvrté vrstvy pak mohou provádět logickou funkci „OR“, která nám sjednotí spolu související jednotlivé konvexní oblasti. Konkrétní neuron čtvrté vrstvy tedy bude aktivní, když vstupní vektor, na který má „reagovat“, bude patřit alespoň do jedné z příslušných oblastí. Obrázek 10 nám toto ukazuje. Tentokrát je v rovině přímek 5, které vytváří dvě konvexní oblasti. Ty jsou znázorněny šedou barvou. Podrobněji je význam jednotlivých vrstev popsán v publikaci [8]. Vícevrstvé sítě typu Perceptron dokáží vyřešit nejen problém XOR, ale v podstatě jakýkoli prostorový úkol. Z předchozího popisu vyplývá, že k tomu stačí tři vrstvy.

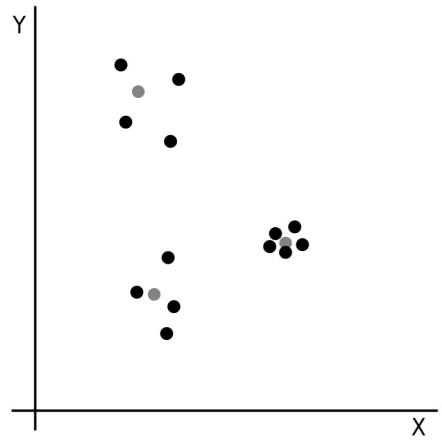
2.7.2 Kohonenovy mapy

Kohonenova mapa nebo také samoorganizační síť byla poprvé popsána v roce 1982. Skládá se pouze ze dvou vrstev. Vstupní vrstvy a výstupní vrstvy. Neurony jsou mezi vrstvami úplně propojené, což znamená že každý neuron vstupní vrstvy je propojen se všemi neurony výstupní vrstvy. Výstupní vrstva je dále uspořádána do nějaké topologické struktury, která navíc zavádí pojem *okolí*. Okolí výstupního neuronu bývá také nazýváno „sousedství “. Nejčastější strukturou výstupní vrstvy bývá jednorozměrná řada nebo dvourozměrná mřížka. V případě dvourozměrné mřížky se může jednat o několik druhů uspořádání neuronů, například čtverecové nebo šestiúhelníkové. V této topologické struktuře určujeme, které neurony spolu sousedí, což je nezbytné pro adaptační (učící) proces. Důležitou součástí pro adaptační proces je tedy ono okolí výstupního neuronu o určitém poloměru R. Toto okolí je množina všech výstupních neuronů, jejichž vzdálenost je od daného výstupního neuronu, ke kterému se okolí vstahuje, menší nebo rovna poloměru R. Způsob měření vzdálenosti mezi neurony je závislý na topologické struktuře výstupní vrstvy. Některé topologie a způsoby měření vzdáleností jsou popsány v publikaci [12].

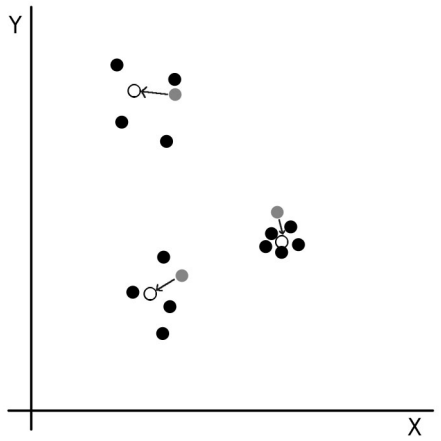
Pro snažší pochopení adaptačního procesu si nejdříve objasníme geometrický význam Kohonenových map.

Tento výklad popíšeme na topologii výstupní vrstvy odpovídající dvourozměrné mřížce. Představme si, že máme spoustu dat, které jsou nějak rozložena v rovině (dvourozměrném prostoru). Tato data nejsou rozložena v rovině rovnoměrně, ale vytvářejí malé hustější oblasti zvané *shluky*. My chceme tyto shluky identifikovat a vybrat pro každý shluk jednoho reprezentanta, který jej bude zastupovat. Chceme tedy vybrat několik takových reprezentantů z naší množiny dat, kteří by co nejlépe zastupovali rozložení dat v prostoru. Na obrázku 11 je ukázka, jak může takové rozložení dat vypadat. Černé tečky představují data a šedé tečky jejich reprezentanty. Reprezentanti nemusejí být součástí množiny dat. Reprezentantem dat může být bod prostoru, který do množiny nepatří, ale je fyzicky nejblíže datům ve shluku. Představuje například těžiště tohoto shluku.

Adaptační proces Kohonenových map patří do skupiny učení bez učitele. Proto se také někdy pro tento typ sítě užívá alternativní název „Samoorganizační síť“ což je obecný název označující neuronové sítě, které se učí bez učitele. Proces adaptace je tedy následující: Předkládáme síti postupně všechny vzory z trénovací množiny. Po předložení každého



Obrázek 11: Rozložení dat v prostoru. Černé body jsou data, šedé jsou reprezentanti.



Obrázek 12: Jeden krok Lloydova algoritmu. Černé body jsou data, šedé jsou reprezentanti a bílé těžiště shluků.

vzoru na vstupní vrstvu probíhá mezi neurony výstupní vrstvy „soutěž“. Vítěz soutěže, tedy neuron, který je nejblíže vstupnímu vektoru, změní své váhové hodnoty spolu s neurony v jeho okolí. Váhové hodnoty vedoucí od vstupů k neuronu ve výstupní vrstvě představují polohu v prostoru tohoto výstupního neuronu. Při adaptaci je tedy potřeba měnit tyto váhy tak, aby se vítězný neuron více přiblížil vstupnímu vektoru a tím jej jasně reprezentoval. Velikost změny vah při adaptaci ovlivňuje také parametr učení α, který nabývá hodnot z rozsahu 0 - 1. Je-li α 1, pak vítězný neuron mění své váhy tak, že se „posune“ do bodu představujícího vstupní vektor. Je-li α 0, pak se vítězný neuron „nehýbe“ vůbec. Je-li hodnota α mezi nulou a jedničkou, přesune se vítězný neuron o tento poměr blíže k bodu reprezentující vstupní vektor. Tady nastává *konflikt stability a plasticity*. Když je parametr učení „velký“, učí se síť rychle novým vzorům, ale zapomíná ty staré. Je-li parametr učení „malý“ učí se síť pomalu, ale pamatuje si i předchozí vzory. Z tohoto důvodů se parametr učení v průběhu adaptace mění. Na začátku je jeho hodnota rovná 1 (nebo téměř 1) a postupně klesá k nule, což zastaví proces adaptace. Změny vah jsou tedy nejvýraznější na začátku učení. Stejně tak se mění i velikost okolí výstupního neuronu. Na začátku bývá okolí „velké“ (až polovina velikosti sítě) a postupně se zmenšuje až na pouhý vítězný neuron.
Matematické vyjádření změny vah je následující:

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + \alpha(x_i - w_{i,j}^{(t)}) \tag{3}$$

Kde:
x_i - je hodnota i-tého vstupního neuronu
$w_{i,j}$ - je hodnota váhy vedoucí od i-tého vstupního neuronu k j-tému výstupnímu neuronu
α - je koeficient učení

(t) - označuje stavy v čase t

„Smyslem Kohonenovy sítě je vystihnout charakter množiny vstupů.“ [7]. Kohonenová mapa tak vlastně odráží statistické vlastnosti vstupních vektorů (bodů). Pokud bychom parametr učení nesnížili až na nulu, ale nechali jej na nějaké „nízké“ hodnotě, bude se síť učit pořád. Kdyby se pak na vstupu objevil vektor, který nespadá do žádné známé skupiny (shluku) může si tak síť, pokud má dostatek neuronů, vytvořit nový. Získali bychom tak síť, která se učí i za „provozu“. Tento typ sítě tedy použijeme např. tam, kde potřebujeme třídit vstupy do skupin. Síť si díky samoorganizaci tyto skupiny vytvoří sama, což je velice přínosné v případech, kdy nemáme tušení, jaké vztahy jsou mezi daty (vstupními vektory). Kohonenovy mapy se v praxi používají například pro zpracování řeči (převod mluveného slova na text), zpracování obrazu (detekce osob podle fotografií), převod ručně psaného textu do elektronické podoby, hledání podobných znaků v neznámých signálech.

2.7.3 Kvantování vektorů učením

Kvantování vektorů učením (anglicky Learning Vector Quantization - LVQ) je založeno na Kohonenově učení. Mluvíme tudíž o Kohonenově mapě, ale rozdíl je v tom, že učení probíhá s učitelem. V trénovací množině tedy bude navíc informace od „učitele“ a tou je příslušnost trénovacího vzoru k určité třídě. Skupiny nebo třídy shluků si tentokráte neurčuje síť sama, ale určíme je my. Řekneme tedy, který výstupní neuron bude reprezentovat tu či onu třídu vstupních vektorů. Adaptace je realizována opět na základě soutěže neuronů výstupní vrstvy, ale liší se úpravou vah. Také se nepoužívá k učení okolí výstupního neuronu, neboť víme předem, který neuron má „zvítězit“. Rozlišujeme tři typy LVQ:

- LVQ 1 - Procházíme trénovací množinu a předkládáme na vstup jednotlivé vzory. V případě, že se jedná o správnou odezvu (zvítězil neuron, který je určen trénovací množinou) provedeme adaptaci stejně jako v případě Kohonenova učení podle vzorce 3. Změnu vah provedeme pouze pro vítězný neuron. V případě chybné odezvy (zvítězil nesprávný neuron) oddálíme vítězný neuron od vstupního vektoru. Vzorec tedy bude vypadat následovně:

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} - \alpha(x_i - w_{i,j}^{(t)}) \tag{4}$$

  V každém kroku tedy posuneme pouze jeden neuron.

- LVQ 2 - tento učící algoritmus pouze vylepšuje předchozí algoritmus (LVQ 1). Posouvá vždy dva nejbližší neurony od vstupního vektoru. Pokud leží vstupní vektor v určité vzdálenosti mezi správným výstupním neuronem a nějakým jiným (reprezentující jinou kategorii), posune se v jednom kroku správný neuron blíže ke vstupnímu vektoru a „špatný“ neuron směrem od vstupního vektoru.

- LVQ 3 - opět pouze vylepšuje předchozí algoritmus (LVQ 2). Přidává jedno pravidlo navíc. Správně klasifikující neurony se navíc pohybují směrem ke vstupnímu vektoru o menší poměrnou část.

Podrobnější vysvětlení lze najít v publikaci [8].

2.7.4 Counter propagation

Síť Counter propagation je dopředná síť skládající se ze dvou různých typů sítí. Tuto síť navrhnul v roce 1986 Hecht-Nielsen. Síť obsahuje tři vrstvy neuronů. První a druhá vrstva tvoří Kohonenovou mapu. Druhá a třetí vrstva tvoří *Grossbergovy hvězdy*. Druhá vrstva neuronů je tedy společná a tvoří zároveň jak výstupní vrstvu Kohonenovy mapy tak vstupní vrstvu Grossbergových hvězd. Spoje vedoucí od všech vstupních neuronů do jednoho neuronu v Kohonenově vrstvě se nazývají *Instar* a spoje vedoucí od jednoho neuronu z Kohonenovy vrstvy do všech Grossbergových jednotek se nazývají *Outstar*.

Grossbergová hvězda je neuronová síť, která má dvě vrstvy neuronů. Vstupní vrstva obsahuje pouze jeden neuron, výstupní vrstva pak několik neuronů. Tato síť (jednotka) nastaví výstup do určitého stavu jen pokud je aktivován vstupní neuron. Aktivace vstupního neuronu tak vytvoří na výstupu konkrétní vektor. Pokud se výstupní vektor liší od požadovaného, provede se adaptace podle Grossbergova pravidla. To je definováno takto:

$$v_j^{(t+1)} = v_j^{(t)} + \alpha(y_j - v_j^{(t)}) \tag{5}$$

Kde:
v_j - je skutečná váha k j-tému výstupnímu neuronu
y_j - je požadovaná váha k j-tému výstupnímu neuronu
α - je koeficient učení (v průběhu klesá až k nule)
(t) - označuje stavy v čase t

V síti Counter propagation je Grossbergova hvězda propojena se všemi neurony v Kohonenově vrstvě. To je sice v rozporu s předem uvedenou definici Grossbergovy hvězdy, ale jelikož je díky soutěživosti neuronů v kohonenově vrstvě aktivní vždy jen jeden neuron, chová se tak síť, jako by v jeden okamžik měla Grossbergova hvězda jen jeden vstupní neuron. Ten je však pokaždé jiný a tím pádem se účastní i jiné váhy. Naučená Grosbergova vrstva pak provádí výběr jednoho vektoru z množiny vektorů, jejíž počet je dán počtem neuronů v Kohonenově vrstvě. Konkrétní výstupní vektory pak udává trénovací množina. Adaptace takové sítě má dvě fáze. V první fázi se trénuje Kohonenová mapa s využitím samoorganizace. Po naučení se zafixují váhy mezi vstupní a Kohonenovou vrstvou a následuje druhá fáze. V této fázi pokračuje adaptace u Grossbergovy hvězdy a mění váhové hodnoty mezi Kohonenovou a Grossbergovou vrstvou.

Tento typ sítě má podstatně vyšší rychlost učení než třeba Perceptronové sítě, ale zase naopak má menší přesnost odezvy. Jedno z použití těchto sítí je například zobrazení $f : R^n \rightarrow R^m$. Reprezentanti vstupů zde mají stejnou pravděpodobnost výběru a výstupní

hodnoty představují průměr funkčních hodnot v okolí těchto reprezentantů. Konkrétním příkladem použití je aproximace funkce sinus na intervalu $< 0, 2\pi >$. Kohonenová vrstva rozdělí vstupní prostor na mnoho malých podintervalů a pro každý podinterval dává pak Grossbergova vrstva průměrnou hodnotu funkce sinus. Dále se pro síť Counter propagation hodí úlohy, které mají povahu vyhledávací tabulky. Vyšší rychlost učení pak můžeme využít, je-li síť integrována do nějakého systému, kde je kladen požadavek na rychlost. Samozřejmě nám nesmí vadit menší přesnost výsledků.

2.7.5 Hopfieldův model

Tento model neuronové sítě navrhli původně McCulloch a Pitts a později jej analyzovali i jiní, včetně Johna Hopfielda. Hopfield použil při analýze stability této sítě analogii k fyzikální teorii magnetických materiálů a tím tento model vešel ve známost. Proto nese síť právě jeho jméno. Hopfieldová síť je navržena jako autoasociativní paměť, která se skládá z neuronů navzájem úplně propojených, tedy každý s každým vyjma sebe samým. Spoje mezi neurony jsou symetrické, což matematicky označujeme pomoci vah $w_{i,j} = w_{j,i}$. Jinými slovy, signály se šíří v obou směrech a váha je pro oba směry stejná. Všechny neurony jsou zároveň vstupní i výstupní. Podobně jako perceptrony mají i neurony této sítě svůj práh a aktivační funkci. Hodnota prahu však bývá nulová. Aktivační funkce je skoková a výstupy neuronů mohou nabývat hodnoty buď binární (0 a 1) nebo bipolární (-1 a 1). Většinou se používají bipolární hodnoty na výstupech. Vstupem do aktivační funkce je vnitřní potenciál daný váženou sumou okolních neuronů. Svojí konstrukci jde Hopfieldův model směrem od biologického vzoru, protože symetrické spoje neuronů nebyly v biologických neuronech pozorovány.

Adaptace Hopfieldovy sítě se provádí dle Hebbova pravidla. To v roce 1949 vyslovil *Donald Olding Hebb* na základě pozorování biologických neuronů. Pravidlo říká, že synaptické spojení mezi dvěma ve stejnou chvíli aktivovanými neurony se posiluje. Podrobnější popis Hebbova pravidla pro adaptaci asociativních sítí je v publikaci [8]. Při adaptaci tedy opět procházíme trénovací množinu, která obsahuje vzory k naučení (bipolární vektory). Předtím ale musíme nastavit všechny váhy spojů na nulu. Teprve pak předkládáme vzory síti. Předložení znamená že nastavíme všechny její neurony na hodnoty vzoru. Po předložení vzoru měníme všechny váhové hodnoty tímto způsobem: Pokud spoj spojuje neurony se stejnou hodnotou, zvýšíme jeho váhu o jedničku. Spojuje-li neurony s opačnými hodnotami, hodnotu váhy snížíme o jedničku. Matematicky vyjádřeno:

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + x_i * x_j \tag{6}$$

Snižování váhy už není zcela v souladu s biologickým vzorem, neboť tam se váhy spojů pouze posilují. Tímto způsobem tedy měníme váhové hodnoty pro všechny trénovací vzory. Po naučení sítě vzorům z trénovací množiny vyjadřuje hodnota váhy spoje rozdíl v počtu vzorů, ve kterých se její neurony shodly svými výstupními hodnotami a počtu vzorů, ve kterých se neshodly. Čili pokud se dva neurony shodnou v pěti vzorech a ve dvou se neshodnou, bude hodnota váhy 3. Když se shodnou jen třikrát a osmkrát ne,

bude hodnota váhy -5.

Aktivní dynamika neboli fáze vybavování probíhá následovně. Výstupy všech neuronů nastavíme na hodnoty vstupního vektoru. Poté vybereme jeden neuron a provedeme výpočet jeho výstupu podle následujícího postupu. Nejprve vypočteme jeho vnitřní potenciál.

$$\xi_j^{(t)} = \sum_{i=1}^{n} w_{j,i} y_i^{(t)} \tag{7}$$

Poté vypočteme nový stav výstupu podle aktivační funkce, která je bipolarní verzí ostré nelinearity.

$$y_j^{(t+1)} = \begin{cases} 1 & \xi_j^{(t)} > 0 \\ y_j^{(t)} & \xi_j^{(t)} = 0 \\ -1 & \xi_j^{(t)} < 0 \end{cases}$$

Tento výpočet postupně opakujeme ve všech neuronech, pořád dokola, dokud se síť nedostane do stabilního stavu. Ten poznáme tak, že se již neobjevují žádné změny ve výstupních hodnotách. Po ustálení sítě jsou výstupní hodnoty neuronů zároveň výstupem sítě. Bylo dokázáno, že pro každý vstup skončí výpočet Hopfieldový sítě ve stabilním stavu po konečném počtu kroků. Problém by nastal pouze tehdy, pokud bychom neprováděli výpočty postupně pro každý neuron zvlášť, ale vypočítali bychom nejdříve všechny výstupní hodnoty najednou a teprve potom bychom změnili výstupy na nově vypočtené hodnoty. Výstup sítě se tak neustále mění dokud se síť nedostane do stabilního stavu. Je to z toho důvodu, že neurony jsou propojené každý s každým a tím pádem se neustále ovlivňují. Jeden neuron svojí váhou excituje jiný neuron, který je zároveň utlumován třetím neuronem. Proces vybavování se dá popsat tzv. energetickou funkcí sítě. Ta má následující tvar:

$$E_{(y)} = \frac{-1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} w_{j,i} y_j y_i \tag{8}$$

Energie sítě během procesu vybavování klesá. Na konci procesu vybavování, kdy je síť v ustáleném stavu se tak nachází v lokálním minimu energetické funkce. Hopfieldová síť se tedy z libovolného počátečního stavu nakonec ustálí v nějakém naučeném vzoru. Tímto vlastně simuluje asociativní paměť. Když porovnáme fáze učení a vybavování Hopfieldovy sítě se sítí Perceptronu, zjistíme že mají opačný charakter. Adaptace Hopfieldovy sítě je jednorázová záležitost v délce odpovídající počtu vzorů, kdežto vícevrstvé Perceptronové sítě potřebují podstatně delší dobu na učení. U vybavovací fáze je tomu naopak.

Praktická aplikace Hopfieldovy sítě je například rozpoznávaní poškozených obrazových vzorů. Dejme tomu že jsme síť naučili poznávat jednotlivá písmena tištěné abecedy. Když například naskenujeme nějaký tištěný dokument, nemusí být jeho kvalita dostatečná a znaky abecedy mohou být špatně čitelné. Hopfieldová síť má tu schopnost, že dokáže rozpoznat i tyto špatně čitelné znaky. Existuje i spojitá verze Hopfieldovy sítě. Ta se využívá

pro řešení optimalizačních problémů jako například „Problém obchodního cestujícího“ [12].

2.8 Flexibilní neuronový strom

Tomuto typu neuronové sítě věnuji samostatnou kapitolu, jelikož je tento typ předmětem této práce.

Jak vyplývá z předchozích popisů jednotlivých typů neuronových sítí, správná volba struktury neuronové sítě se vysoce podílí na jejím celkovém výkonu a schopnostech. Vzájemné působení dvou nebo více neuronů mezi sebou je dáno existencí spojů mezi nimi. To, jestli určité neurony spolu budou nebo nebudou propojeny, je dáno právě samotnou strukturou neuronové sítě. V závislosti na konkrétním problému může být například vhodné mít více než jednu skrytou vrstvu, dopředné nebo zpětné spoje mezi neurony, nebo v některých případech dokonce přímé propojení některých vstupních neuronů přímo do výstupní vrstvy. Tradiční přístup k neuronovým sítím volí statické struktury navrhované manuálně. Výsledný výkon takto navržené neuronové sítě je tudíž ovlivněn zkušenostmi návrháře. Proto bylo realizováno mnoho pokusů o automatický návrh vhodné struktury neuronových sítí. Jedny z prvních metod automatického návrhu struktury používaly tzv. *konstruktivní* a *prořezávací* algoritmy. Ty však měly určité problémy.

2.8.1 Neuronový strom

Novější přístup, inspirovaný práci Byoung-tak Zhanga, je využití struktury stromu jako modelu neuronové sítě. Zhang ve své práci představil metody evoluční indukce řídkých neuronových stromů. Model flexibilního neuronového stromu můžeme vytvořit a vyvíjet na základě předdefinované sady instrukcí (operátorů). V tomto typu struktury sítě jsou povoleny spoje přes více vrstev, různé aktivační funkce pro jednotlivé neurony v rámci jedné sítě a výběr vstupů, které budou k danému neuronu připojeny [1].

Hledání vhodné struktury neuronového stromu může být provedeno pomocí různých evolučních algoritmů založených na stromové struktuře. Naučení sítě, v tomto případě spíše doučení nebo „doladění “ parametrů sítě může být provedeno optimalizačními algoritmy. Máme tak vlastně dvě různé optimalizace. Jedna slouží pro úpravu struktury stromu a druhá pro nalezení vhodných parametrů stromu. Metody pro nalezení optimálního flexibilního neuronového stromu pro konkrétní problém střídají obě optimalizace. Hledání vhodného řešení se nám tak rozpadá na dvě základní fáze. Na začátku se vygenerují náhodné stromové struktury s náhodně nastavenými parametry (váhy, prahy, atd.) V první fázi se evoluční algoritmus nejdříve snaží vylepšit tyto struktury. Jakmile jsou nalezeny lepší struktury neuronových stromů, pokračuje se s dolaďováním parametrů (druhá fáze). Poté se opět vylepšuje struktura a následně vylaďují parametry. Obě fáze se neustále opakují, dokud se nenalezne vyhovující řešení nebo se nepřekročí časový limit. Tento iterativní proces se nazývá vývojový proces neuronového stromu. Dvoufázová optimalizace je jedním ze základních rozdílů ve srovnání s klasickým přístupem k

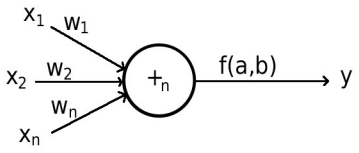
neuronovým sítím, kde struktura byla pevně dána a optimalizovaly se pouze parametry sítě. Za výhodu automatického návrhu struktury sítě však platíme zvýšeným nárokem na výpočetní výkon. Tuto nevýhodu lze však částečně odstranit použitím paralelního výpočtu. Paralelní algoritmus pro vývoj FNT (z angličtiny Flexible Neural Tree) je popsán v publikaci [9].

Zakódování neuronového stromu
V závislosti na tom, jaké algoritmy budou použity v procesu optimalizace, je potřeba zvolit vhodnou formu zakódování neuronového stromu. Každý algoritmus totiž pracuje s odlišnými objekty. Například evoluční algoritmy pracují s DNA řetězci. Tyto řetězce se na nejnižší úrovni skládají z genu. Použijeme-li tedy evoluční algoritmus k optimalizaci struktury stromu, je potřeba před touto operaci neuronový strom transformovat do podoby DNA řetězce.

Flexibilní neuronový operátor
Podobně jako se generují gramatiky můžeme generovat také neuronové stromy. Použijeme k tomu množinu funkcí F (neterminály) a množinu terminálů T. V tomto pojetí jsou terminály listy stromu a reprezentují jednotlivé vstupy neuronového stromu. Funkce představují uzly stromu s n potomky a reprezentují jednotlivé neurony. Množinu operátorů pak můžeme vyjádřit takto:

$$S = F \bigcup T = \{+_2, +_3, \ldots, +_N\} \bigcup \{x_1, x_2, \ldots, x_n\} \tag{9}$$

Kde:
$+_i (i = 2, 3, \ldots, N)$ - představuje operátor nelistového uzlu stromu (neuronu) s i argumenty.
$(x_1, x_2, \ldots, x_n)$ - představuje operátor listového uzlu stromu (vstupu), který nemá žádné argumenty.

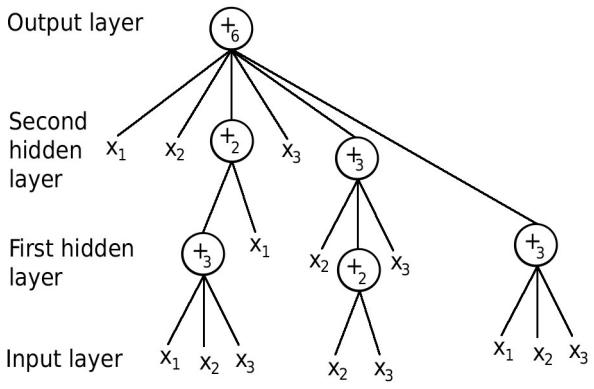


Obrázek 13: Flexibilní neuronový operátor

Když se pak při konstrukci neuronového stromu vybere neterminální operátor, řekněme například $+_3$, postupuje se takto:

- vytvoří se neuron a k němu tři spoje, které budou propojovat daný neuron s jeho potomky

- vygenerují se tři náhodná reálná čísla, která se použijí jako hodnoty vah pro spoje neuronu
- dále se vygenerují další dvě náhodná reálná čísla, která se použijí jako hodnoty parametrů a a b flexibilní aktivační funkce



Obrázek 14: Typická reprezentace neuronového stromu

Flexibilní aktivační funkce je dána vztahem:

$$f(a, b, net) = e^{-(\frac{net-a}{b})^2} \tag{10}$$

Kde:
a a b - jsou parametry
net - je celková excitace

Celková excitace se vypočítá:

$$net = \sum_{j=1}^{n} w_j * x_j \tag{11}$$

Kde:
x_j - jsou vstupy neuronu
w_j - jsou váhy spojení

Výstupem flexibilního neuronu je výstup aktivační funkce. Výstup celého neuronového stromu se spočítá rekurzivně zleva doprava a do hloubky.

Funkce zdatnosti
Aby bylo možno použít evoluční algoritmus pro optimalizaci struktury neuronového stromu, musíme stanovit, jak se bude vypočítávat tzv. funkce zdatnosti (Fitness function). Zdatnost říká, jak moc je daný strom použitelný pro konkrétní úlohu. Je to realné číslo

ohodnocující „výkonnost“ každého stromu v populaci. Funkci zdatnosti vypočítáme například jako průměrnou kvadratickou chybu (RMSE - root mean square error) přes celou trénovací množinu. Můžeme jí také počítat jako MSE (mean square error) nebo dokonce přísněji jako maximální chybu. Nejprve se tedy na funkci zdatnosti můžeme dívat jako na chybu (odchylku skutečné hodnoty od požadované hodnoty) výstupu neuronového stromu. Čím menší chyba, tím lepší výkon. Pak, i když to není nutné, můžeme ještě dále posuzovat velikost stromu měřenou jako počet neuronů ve stromu. Pokud by tedy dva nebo více stromů měly stejnou chybu, vybere se ten, který má menší strukturu. Funkce zdatnosti vyjádřená jako MSE má tvar:

$$Fit(i) = \frac{1}{P}\sum_{j=1}^{P}(y_1^j - y_2^j)^2 \tag{12}$$

RMSE má pak tvar tento:

$$Fit(i) = \sqrt[2]{\frac{1}{P}\sum_{j=1}^{P}(y_1^j - y_2^j)^2} \tag{13}$$

Kde:
$Fit(i)$ - je fitness hodnota i-tého stromu v populaci
P - je celkový počet trénovacích vzorů
y_1^j - je požadovaná hodnota výstupu j-tého trénovacího vzoru
y_2^j - je skutečná hodnota výstupu j-tého trénovacího vzoru

Vývoj neuronového stromu
Při vývoji neuronového stromu, řešícího konkrétní problém, optimalizujeme strukturu i parametry zároveň. Nelze optimalizovat nejdříve jednu část a potom tu druhou. Vychází to z principu přístupu k problému, kterým je „hledání“. Při tomto hledání vzniká potřeba porovnávat mezi sebou vždy dva jedince (neuronové stromy) abychom mohli určit, který z nich je pro konkrétní problém vhodnější. Jedince porovnáváme podle jejich fitness funkce, která byla popsána výše. Hodnotu fitness funkce vypočítáme z výstupní hodnoty neuronového stromu, která je závislá jak na parametrech, tak na struktuře neuronového stromu. To je důvod, proč probíhají obě optimalizace současně.

Hledání optimální nebo téměř optimální struktury neuronového stromu může být prováděno například pomocí *Genetic programming* (GP), *Probabilistic Incremental Program Evolution* (PIPE), *Gene Expression Programming* (GEP), apod. Pro optimalizaci parametrů zase můžeme použít *Genetic Algorithm* (GA), *Evolution Strategy* (ES), *Evolution Programming* (EP), *Particle Swarm Optimization* (PSO), *Simulated Annealing* (SA) nebo gradientní metody. Při tomto přístupu se obě optimalizace navzájem ovlivňují a proto je dobré držet se několika zásad. Jestliže máme například špatnou strukturu sítě, nemá cenu se dlouze zabývat jejím naučením. K požadovanému výsledku by to nevedlo. Proto provádíme optimalizaci parametrů jen po určitý počet kroků a potom jí vystřídáme za optimalizaci struktury. Máme-li nalezenou téměř vhodnou strukturu, nemá cenu tuto konkrétní strukturu dále vylepšovat abychom nepřišli o potenciální vhodné řešení. Abychom v průběhu

procesu neztratili nejlepší jedince, které jsme prozatím nalezli, můžeme použít techniku elitářství. Ta spočívá v tom, že v průběhu procesu udržujeme určitý počet jedinců s nejlepší fitness hodnotou, které označujeme jako elitu. U elitních jedinců pak nehledáme lepší strukturu, případně je ani neučíme.

2.9 Využití neuronových sítí

Největším přínosem neuronových sítí je jejich schopnost generalizovat. Přestože při učení předložíme síti pouze několik příkladů se správnou odpovědí, dokáže síť odpovídat i na takové vstupní hodnoty, které se neučila. Je to proto, že síť realizuje spojité zobrazení z prostorů vstupů do prostoru výstupů. Tím že předložíme síti ve fázi učení několik příkladů, určíme vlastně několik diskrétních hodnot v prostoru vstupů. Síť proloží tyto diskrétní hodnoty hyperplochou a takto pak realizuje spojitost zobrazení. Taková hyperplocha nemusí vždy představovat ideální proložení diskrétních hodnot, což znamená že odpovědi pak nejsou přesné, ale pro člověka jsou tyto odpovědi „rozumné“. Toto chování neuronových sítí se dá využít například takto: máme-li nějakou množinu dat, o kterých víme, že jsou nějak závislá, ale neumíme tuto závislost najít, předložíme tyto data síti a ona nám tu závislost najde.

Dále si ukážeme několik příkladů konkrétního použití neuronových sítí.

- Vyhledávání informací (Information retrieval) - disciplína, která se zabývá metodami hledání relevantních informací z rozsáhlého množství dat, textů, článků a dokumentů. Experimenty potvrdily, že tradiční metody vyhledávání informací označují za relevantní pouze zlomek dokumentů z těch, které jsou pro člověka skutečně relevantní [10]. Důvody jsou následující:
  - striktně matematické modely na základě podobnostních funkcí, které ne zcela odpovídají lidskému chápání podobnosti,
  - neschopnost pracovat s kontextem. Matematický model nerozlišuje míru významnosti termů a jejich kombinací,
  - tradiční systémy vyhledávání informací předpokládají homogenní datové zdroje, ačkoli uživatel očekává heterogenní odpověď.

  Neuronové sítě mohou poskytnout lepší přístup k tomuto problému, protože zpracovávají data paralelně a distribuovaně a představují tak tolerantní a adaptivní systémy.
- Klasifikace - je jedna ze základních oblastí neuronových sítí. Hojně je zastoupena i v medicíně [5].
  - zpracování obrazu - rozpoznávání jednoduchých vzorů například při analýze v radiologii, pozitronovo-elektronové tomografii a sonografii,
  - zpracování signálu - uplatnění v neuronové analýze EKG vln v kardiologii a EEG vln v neurofyziologii,

  - zajímavé uplatnění je také v klinických a experimentálních farmakokinetických a farmakodynamických studiích. Pro analýzu nelineárních údajů jsou mnohem flexibilnější než tradiční polyexponenciální farmakokinetické modely.

- Predikce - zejména časových řad. Je asi jedna z nejrozšířenějších oblastí aplikací umělé inteligence, do níž neuronové sítě patří. V ekonomice se například používá pro předpověď vývoje směnných kurzů, cen akcií na burze a podobně. Zde je základem pro předpověď víra, že existuje funkce, která na základě znalosti x předcházejících hodnot dokáže předpovídat následující vývoj.

- Komprese dat - základem je třívrstvá síť (např. typu Perceptron), jejíž vstupní a výstupní vrstva má stejný počet neuronů. Prostřední vrstva pak musí mít neuronů méně než vrstva vstupní a výstupní. Síť se učí identitě. To znamená, že se na vstup přivede určitý vektor dat a výstup musí vydat stejný vektor. Jelikož má prostřední vrstva méně neuronů než vstupní vrstva, je síť nucena transformovat vícerozměrná vstupní data do méněrozměrného prostoru prostřední vrstvy. Po naučení se síť rozdělí na dvě části. První část (vstupní a prostřední vrstva) tvoří kompresní část. Druhá část sítě (prostřední a výstupní vrstva) tvoří dekompresní část. Vyvstává zde problém, jak zamezit ztrátovosti komprese, ale u některých aplikací, jako například digitální fotografie, to nemusí vadit.

Jedním z problémů v praxi při konkrétním použití neuronové sítě je také vhodná volba trénovací množiny. Špatně zvolená trénovací množina může snížit kvalitu výsledků neuronové sítě. Problém může nastat například s výběrem konkrétních trénovacích vzorů. Viz. např. publikace [7] kapitola 3.4.6, kde se popisuje trénování neuronové sítě pro rozpoznávání objektů na sonaru. Někdy také může být složité převést reálný problém do formy vektoru trénovacího vzoru.

3 Přehled současného stavu poznání

V roce 1997 napsali *Byoung-Tak Zhang*, *Peter Ohm* a *Heinz Mühlenbein* publikaci nazvanou *Evolutionary Induction of Sparse Neural Trees* (Evoluční indukce řídkých neuronových stromů) [16]. V úvodu této publikace se zabývají vícevrstvými perceptronovými sítěmi a neuronovými sítěmi vyšších řádů. Popisují výhody a nevýhody obou typů sítí, jako například větší schopnost učení sítí vyšších řádů, ale zároveň horší zobecňování. Dále pak popisují, jak lze pomocí řídkých neuronových stromů odstranit nevýhody těchto sítí a zároveň ponechat jejich výhody. Ve své práci navrhují nejen vhodnější model neuronové sítě, ale také způsob, jak pro tento model najít vhodnou strukturu a optimální nastavení parametrů. Pátá kapitola jejich publikace ukazuje použití nového modelu pro předpověď časových řad.

3.1 Řídké neuronové stromy

Řídké neuronové stromy vycházejí z modelu vícevrstvé neuronové sítě. Jeden z problémů klasické vícevrstvé sítě je tzv. „kombinační exploze“ parametrů s rostoucím počtem vstupů. Proto autoři navrhují strukturu stromu a odstraňují podmínku, aby všechny vrstvy byly mezi sebou úplně propojeny. To povoluje neexistenci některých spojů mezi vrstvami, což činí takovou strukturu „řidší “. Také vycházejí z praktických zkušeností, že ne každý vstup je stejně důležitý a tudíž nutný pro správnou činnost neuronové sítě. Na základě toho je povoleno používat jen některé vstupy a propojení vstupů přes několik vrstev, dokonce přímo do výstupní vrstvy. Aby zamezili dalšímu problému, se kterým se potýkají neuronové sítě vyšších řádů, jakým je nekontrolovatelný růst velikosti stromu, upravují fitness funkci tak, aby „penalizovala“ složitost struktury. Neurony, ze kterých se stromy skládají, jsou dvojího typu a jsou nazývany neuronovými jednotkami. Jeden typ neuronů jsou tzv. *sigma jednotky*, které provádějí vážený součet vstupů, druhý typ neuronů jsou *pi jednotky*, které provádějí vážený součin vstupů. Tyto jednotky bývají označovány symboly řecké abecedy Σ a Π.

3.2 Flexibilní neuronové stromy

Na základě těchto poznatků začaly vznikat další publikace citující dílo Zhanga a jeho kolegů. Jednou z nich je práce *Tree-Structure Based Hybrid Computational Intelligence*, kterou vytvořili *Ajith Abraham* a *Yuehui Chen* [1]. Druhá část této práce nese název *Flexible Neural Trees* a zabývá se právě řídkými neuronovými stromy. Zde jsou tyto stromy pojmenovány jako Flexible Neural Trees (Flexibilní neuronové stromy). Sady terminálů a funkcí jsou nazvány flexibilními neuronovými operátory. Stromy se generují na základě předdefinované sady těchto operátorů. Pokaždé, když se při konstrukci stromu vybere nelistový operátor (funkční), vygenerují se náhodně hodnoty jeho parametrů. Ke konstrukci stromu jsou v této práci používány pouze sumační neurony a jako aktivační funkce je vybrána *radiální báze*. Funkce zdatnosti je popisovaná jako střední kvadratická chyba (MSE) nebo odmocnina ze střední kvadratické chyby (RMSE). Práce ukazuje použití flexibilních neuronových stromů na mnoha příkladech, jako jsou aproximace funkce, identifikace

nelineárních systémů, detekce narušení, předpověď časových řad, předpověď směnných kurzů, rozpoznávání tváře, klasifikace rakoviny, klasifikace proteinů, apod. Některé výsledky porovnává s jinými přístupy k řešení problémů. Práce ukazuje, že použití flexibilních neuronových stromů dává ve všech případech přesnější výsledky než doposud používané metody. Tato práce je zároveň jakýmsi souhrnem několika předchozích publikací, které vznikaly postupně ve spolupráci autorů a jejich dalších kolegů, kterými jsou například *Shuyan Jiang* a *Lizhi Peng*. Jednotlivé práce jsou později mnohokrát citovány dalšími lidmi zabývajícími se podobnou oblastí. Flexibilní neuronový strom si našel místo i v oblasti dolování dat.

3.3 Nové publikace a výsledky

V roce 2011, popsali *Lizhi Peng*, *Bo Yang*, *Lei Zhang* a *Yuehui Chen* možnost paralelizace vývojových algoritmů pro flexibilní neuronový strom [9]. Zabývají se myšlenkou, že pokaždé když je vygenerována nová populace jedinců, může být prováděno jejich ohodnocování paralelně. Také proces optimalizace parametrů jedince (učení sítě) je možno paralelizovat. V tomto případě se jako optimalizační algoritmus používá *Particle Swarm Optimization* (PSO), takže paralelizace spočívá v ohodnocování populace parametrů. Toto vylepšení účinnosti modelu flexibilního neuronového stromu autoři nazývají *Parallel evolving algorithm for FNT* (PE-FNT) neboli paralelní vývojový algoritmus pro flexibilní neuronový strom.

V květnu 2012 se objevuje práce nazvaná *Flexible Neural Trees for Online Hand Gesture Recognition using Surface Electromyography* v odborném časopise *JOURNAL OF COMPUTERS* [14]. Volně přeloženo, flexibilní neuronové stromy pro rozpoznávání gest ruky v reálném čase pomocí povrchové elektromyografie. Autory této práce jsou *Ajith Abraham*, *Yina Guo*, *Qinghua Wang* a *Shuhua Huang*. FNT by měly usnadnit a automatizovat vyhodnocování signálů z přístrojů pro povrchovou elektromyografii. Použití flexibilních neuronových stromů pro rozpoznávání gest ruky je upřednostňováno před klasickými neuronovými sítěmi z důvodu rychlejší odezvy na podněty (vstupy). Toho je dosahováno schopností FNT vybrat si automaticky potřebné vstupy. V tomto případě tak není zapotřebí složitě určovat, které ze signálů jsou důležité pro správnou detekci. Při této práci bylo dosaženo rozpoznávání šesti různých gest v reálném čase s úspěšností 97,5%. Navíc se zde potvrzuje, že s automaticky vybranými vstupy má neuronový strom lepší přesnost a dobrou schopnost zobecňování.

Jiná práce z roku 2012 nesoucí název *Small-time scale network traffic prediction based on flexible neural tree* ukazuje použití FNT pro předpovídání síťového provozu v malých časových rozsazích [4]. Pro optimalizaci struktury neuronového stromu tentokráte autoři volí GP, oproti dříve často používanému optimalizačnímu algoritmu PIPE. Funkci zdatnosti tentokráte počítají jako normalizovanou střední kvadratickou chybu (NMSE). Autoři ve své práci opět porovnávají FNT s klasickými dopřednými neuronovými sítěmi. V tomto případě poukazují na velký problém zvolit vhodnou strukutru klasické neuronové sítě, její slabost v pomalé konvergenci, možné přeučení a tím snížení schopnosti

zobecňovat. Také poukazují na problémy s uváznutím v lokálním minimu. Jelikož se pro předpovídaní síťového provozu již klasické neuronové sítě používají, mohli porovnat práci, kterou je potřeba vykonat při návrhu řešení. Použití FNT „drasticky“ změnilo problém v nalezení vhodné struktury a optimalizaci parametrů v pozitivním směru. Výhody, které má FNT oproti klasickým neuronovým sítím, uvádějí tyto:

- vstupy, výstupy a struktura sítě pro konkrétní problém nemusejí být předem navržené (FNT si je najde sám),
- struktura FNT je obvykle mnohem jednodušší a má lepší zobecňování,
- evoluční algoritmus automaticky zajistí výběr vhodných vstupů a může upřednostnit menší struktury.

V roce 2013 se objevuje studie *Yu Wanga*, která se zabývá vylepšením optimalizačních algoritmů pro vývoj FNT [15]. Pro optimalizaci struktury navrhuje použít vylepšenou verzi genetického programování zvanou *Multi expression programing* (MEP). Ta spočívá v odlišnosti používání chromozomu. Algoritmus původního GP kóduje jednoho jedince (řešení problému) do jednoho chromozomu. Novější přístup kóduje do jednoho chromozomu více jedinců. Jedinec s nejlepší zdatnosti (Fitness) pak reprezentuje chromozom. Dále shledal často používaný algoritmus optimalizace prametrů PSO náchylným k uváznutí v lokálním optimu. Z tohoto důvodu navrhuje určité změny, které algoritmus vylepšují. Vylepšenou verzi algoritmu nazývá *Regional selection particle swarm optimization* (RSPSO).

Zajímavá je i práce nazvaná *Evolving Flexible Beta Basis Function Neural Tree for Nonlinear Systems*. Ta ukazuje použití *Beta funkce* jako aktivační funkce neuronů. Beta funkce používá čtyři parametry a je oproti standardně používané gausové funkci (typ fukce radiální báze) se dvěma parametry flexibilnější a univerzálnější. Tento model je nazván *Flexible Beta Basis Function Neural Tree* (FBBFNT). Model samotný se liší pouze aktivační funkci a názvoslovím. Pro optimalizaci struktury je použito *Extended Immune Programming* (EIP) jelikož má vyšší konvergenční schopností než GP. Umožňuje totiž docela rychle nalézt úspěšná řešení i s menší populaci. Pro optimalizaci parametrů byl zvolen algoritmus *Hybrid Bacterial Foraging Optimization Algorithm* (HBFOA). Je to upravená verze algoritmu (BFOA), která k základnímu algoritmu přidává prvky *Differential evolution* (DE) a (PSO). Úprava spočívá ve změně výpočtu délky kroku bakterie při přemisťování. Do výpočtu se započítává také zdatnost aktuální bakterie a zdatnost bakterie s dosud nejlepší zdatností. Délka kroku bakterie se tak přizpůsobuje aktuálním podmínkám a je v průběhu procesu proměnná. Očekává se , že proměnná délka kroku oproti pevné délce přinese zlepšení v podobě rychlejší konvergence.

3.4 Přínos FNT

Z předchozích publikací vyplývá, jaký přínos má model flexibilního neuronového stromu. Především řeší problém vhodné volby struktury neuronové sítě, čímž ulehčuje práci návrhářům řešení. Dále odstraňuje některé slabosti klasických neuronových sítí jako jsou

velká komplexita struktury a tím pomalejší odezvy a delší učení (optimalizace parametrů). Schopnost výběru vhodných vstupních proměnných a jejich možnost propojení i do vyšších vrstev se příznivě projevuje na celkovém výkonu neuronového stromu. Přes mnohé popisované výhody však FNT nemůže nahradit veškeré typy klasických neuronových sítí. Jeho předností jsou především na poli predikce časových řad a klasifikace.

3.5 Aplikace pro práci s FNT

V průběhu této práce se mi nepodařilo nalézt žádnou dostupnou aplikaci nebo knihovnu, která by umožňovala práci s FNT. Pro práci s klasickými neuronovými sítěmi existuje celá řada různých knihoven, modulů i celých aplikací. Jednou z takovýchto aplikací je opensource projekt *Encog*, rámec pro podporu strojového učení společnosti Heaton Research. Ten však podporu pro neuronovou síť typu FNT, alespoň zatím, nemá. V lednu roku 2012 však bylo vytvoření podpory pro FNT přidáno jako úkol pro jednu z příštích verzí.

4 Implementace knihovny FNT

Knihovna pro vývoj flexibilního neuronového stromu je implementována v jazyce C# za pomoci vývojového prostředí Microsoft Visual Studio 2010.

4.1 Požadavky na fukcionalitu

- Základním požadavkem je automatické vytvoření neuronového stromu (případně více stromů) pro řešení konkrétního problému, popsaného v kapitole *Flexibilní neuronový strom*. Uživatel knihovny předloží jako vstup informace popisující konkrétní problém, který chce řešit, a knihovna provede automatické nalezení vhodného řešení tohoto problému ve formě jednoho nebo několika neuronových stromů.
- Model neuronového stromu by měl být oddělen od algoritmů pro jeho nalezení, aby se dal jednoduše používat.
- Nalezené řešení (jeden nebo více neuronových stromů) by mělo jít uložit do souboru, aby bylo možno jeho uchování a opětovné použití. Formát tohoto souboru by měl být nezávislý na prostředí, aby bylo možno jej zpracovávat kterýmkoli jiným programem.

4.2 Analýza požadavků

- Aby bylo možno nalezené řešení ukládat do souborů a také je z něj načítat, bude knihovna obsahovat dvě veřejné metody pro tuto práci. Formát souboru bude standardní XML, jelikož je to jeden z nejrozšířenějších formátů a má podporu ve většině aplikací.
- Jelikož je neuronový strom typ dopředné neuronové sítě podobný síti Perceptronů, bude jako optimalizační algoritmus použita učící metoda Back-propagation, popsaná v kapitole 2.6 *Učící algoritmus zvaný Back-propagation*. Zde však použijeme její parametrickou verzi, která modifikuje nejen váhy spojů, ale i prahy a strmostí neuronů. Mělo by to zajistit lepší konvergenci.
- Struktura neuronové sítě se dá popsat maticí spojení. Tato matice se pak dá převést na jednodimenzionální vektor spojení, vyjádřený čísly 0 a 1. Na takový vektor se dá dívat podobně jako na DNA řetězec. To umožňuje pro optimalizaci struktury použít algoritmus genetického programování, popsaný v sekci 4.3 *Princip optimalizace pomoci GP*, takže jej použijeme.
- Protože oba zvolené optimalizační algoritmy mají několik parametrů, které zásadně ovlivňují jejich chování a také výsledné řešení, bude knihovna obsahovat možnost tyto parametry ovlivnit uživatelem.
- Aby model neuronového stromu pokrýval co nejširší oblast použití, budou neurony disponovat třemi různými aktivačními funkcemi. Funkce *Radiální báze* a *Standardní*

sigmoida umožňují výstupní hodnoty v intervalu $\langle 0; 1 \rangle$. Pro výstupní interval $\langle -1; 1 \rangle$ bude použita funkce *Hyperbolický tangens*.

- Učení neuronové sítě algoritmem back-propagation může zabrat delší čas. Záleží to především na velikosti sítě, na velikosti maximální chyby, kterou požadujeme, a na dalších parametrech algoritmu. Když k tomu přičteme čas, potřebný pro nalezení vhodé struktury sítě, je velmi pravděpodobné, že celková doba procesu hledání vhodného řešení může zabrat řádově i dny. Aby bylo možné sledovat, v jaké fázi se proces vývoje neuronového stromu nachází, bude knihovna disponovat několika událostmi. Pokud by knihovna byla součástí nějaké jiné aplikace, může tak prostřednictvím událostí s touto aplikaci komunikovat.

- Obecně může mít neuronová síť jeden nebo více vstupů a také jeden nebo více výstupů. Flexibilní neuronový strom, popsaný v kapitole 2.8, má ale jen jeden výstup. Více výstupů zajistíme tak, že vytvoříme více neuronových stromů, které budou mít společné vstupy. Z vnějšího pohledu bude tato koncepce vypadat jako neuronová síť o několika vstupech a několika výstupech.

4.3 Princip optimalizace pomoci GP

GP - genetic programing, neboli genetické programování, je metoda strojového učení používající evoluční algoritmy, které jsou založeny na metodách podobných biologické evoluci. Obecně se snaží vytvořit a vylepšit počítačový program, který řeší danou úlohu. Existuje několik variant genetického programování, ale základní princip je stejný nebo alespoň velmi podobný. Genetické programování používá termíny z oblasti evoluční biologie jako například:

- *Jedinec* - vhodně zakódovaný počítačový program. V našem případě zakódovaný neuronový strom.

- *Zdatnost* - (fitness) je výsledek funkce zdatnosti (fitness function), která každého jedince ohodnotí číslem, udávajícím jeho schopnost řešit daný úkol.

- *Populace* - množina jedinců, kteří jsou zpracováváni v jednom kroku procesu.

Cílem je nalézt jedince s nejvyšší zdatnosti, čili v našem případě neuronový strom s nejmenší odchylkou skutečné hodnoty výstupu od požadované hodnoty. Jednotlivé kroky algoritmu jsou:

1. Inicializace - vytvoření prvotní populace. Obvykle to jsou náhodně vygenerovaní jedinci.

2. Ohodnocení - přiřazení hodnoty podle funkce zdatnosti všem jedincům.

3. Test - zjistí se, jestli některý z jedinců splňuje podmínky pro ukončení algoritmu hledání. Pokud ano, algoritmus se ukončí. Pokud ne, pokračuje se následujícím bodem.

4. Vytvoření nové populace - náhodně se vyberou jedinci s vyšší zdatností a z nich se vygenerují noví jedinci. Pak se pokračuje bodem 2.

Pro vytváření nové populace se používají tyto operátory:

- reprodukce - zkopíruje jedince
- křížení - vymění části jedinců mezi sebou
- mutace - náhodně změní malou část jedince

Velikost populace se udržuje na stále stejné hodnotě. To znamená, že při vytváření nové populace se z ní musí odstranit tolik jedinců, kolik jich bylo vygenerováno nových.

Tento algoritmus může být zdlouhavý, případně náročný na výpočetní výkon a dokonce se v rozumném čase ani nemusí nejlepší řešení nalézt. Dokáže však nalézt řešení více či méně se přibližující optimálnímu řešení.

4.4 Návrh struktury knihovny

Knihovna bude obsahovat základní jmenný prostor *FlexibleNeuralTree*. V tomto jmenném prostoru budou vnořeny ještě tři další jmenné prostory, které budou logicky dělit některé knihovní třídy.

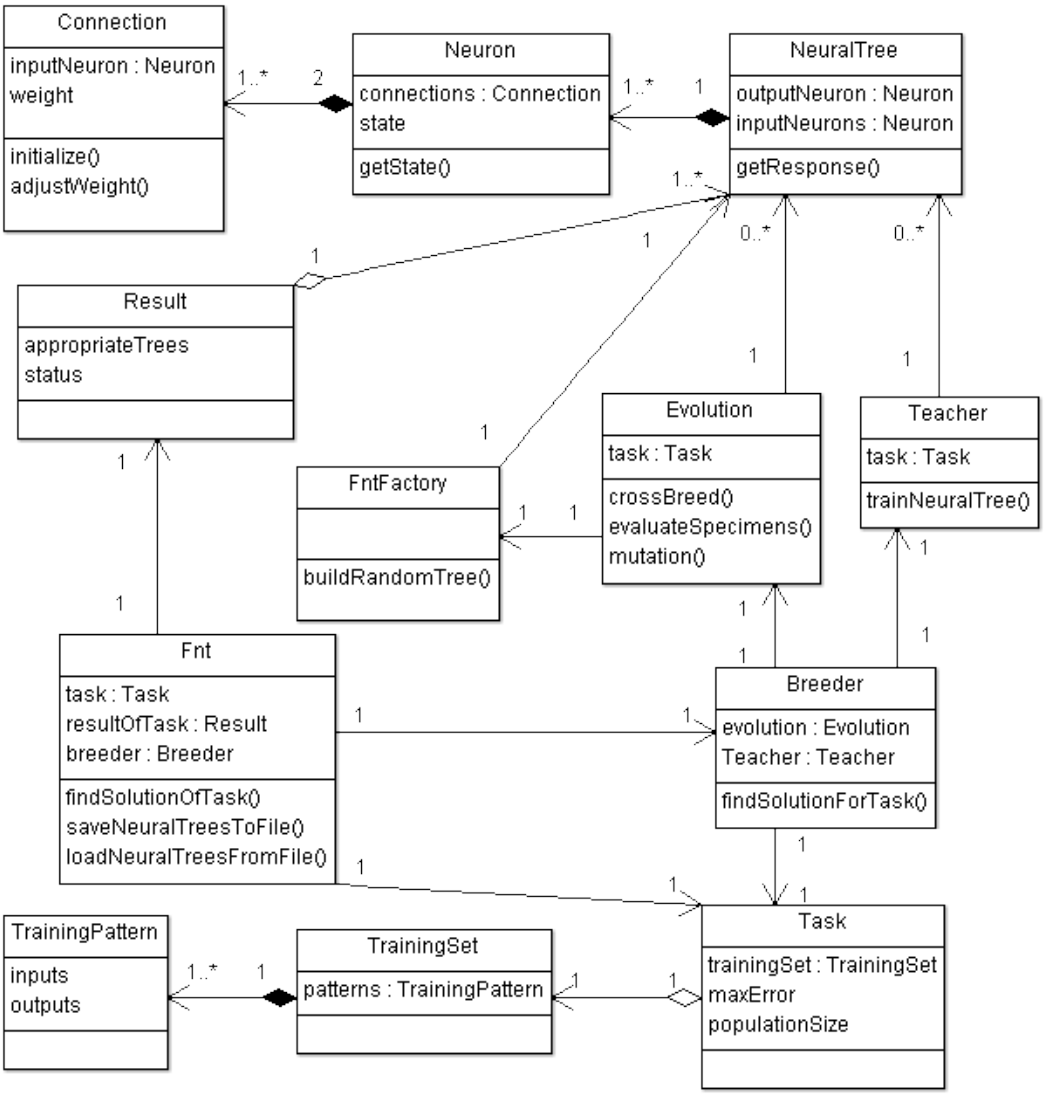
Jmenný prostor *TreeModel* bude obsahovat třídy *Neuron*, *Connection* a *NeuralTree*. Tyto třídy budou představovat implementaci flexibilního neuronového stromu a budou mít všechny potřebné metody pro základní práci.

Jmenný prostor *Workers* bude obsahovat třídy *Breeder* a *Teacher*. Ty budou implementovat zmíněné optimalizační algoritmy a budou zodpovědné za nalezení vhodné struktury stromu a jeho naučení řešení konkrétního problému.

Jmenný prostor *Utilities* bude obsahovat podpůrné třídy pro vnitřní potřebu, jako například *Calculator* pro výpočty funkcí, *FntFactory* pro vytváření neuronových stromů atd.

Ostatní podpůrné třídy, se kterými se dostane do kontaktu uživatel knihovny, budou přímo v hlavním jmenném prostoru. Třída *Task* bude sloužit jako zadání, které bude obsahovat popis problému a nastavení jednotlivých parametrů potřebných pro nalezení vhodného řešení. Pro popis problému bude sloužit třída *TrainingSet*. Ta bude představovat trénovací množinu. Trénovací množina vlastně plně popisuje konkrétní problém, pro který je potřeba nalézt řešení. Nalezené řešení v podobě jednoho nebo více neuronových stromů pak bude představovat třída *Result*. Ta bude, kromě řešení samotného, ještě obsahovat údaje o proběhlém procesu, jako například čas, který byl potřebný pro zpracování úkolu, počet vhodných nalezených řešení, jestli bylo dosažené maximální povolené chyby či nikoli atd. Třída, která celý tento model propojí a synchronizuje, bude mít název *Fnt*. Bude obsahovat jednu metodu, které se předá jako argument třída *Task*. Ta zajistí nalezení

vhodného řešení a vrátí třídu *Result*, která bude obsahovat řešení problému. Dále bude třída *Fnt* obsahovat metody pro uložení a načtení nalezeného řešení. Některé základní třídy a jejich vzájemné vztahy ukazuje třídní diagram na obrázku 15.



Obrázek 15: Třídní diagram - obsahuje pouze nejdůležitější třídy a základní vztahy

4.5 Implementace knihovny

Implementaci knihovny popíšu podle jednotlivých logicky souvisejících skupin tříd z pohledu uživatele. Takto to bude pochopitelnější oproti chronologickému postupu popisu, jak třídy postupně vznikaly. Věnovat se budu převážně těm detailům, které jsou důležité

pro efektivní používání knihovny. Porozumění základnímu principu fungování knihovny pomůže zlepšit orientaci v parametrech, kterými lze proces vývoje neuronového stromu ovlivnit.

Pro veškeré pojmenování tříd, třídních členů, vlastností, metod atd. jsem použil anglický jazyk. Angličtina je rovněž použita i v komentářích.

4.5.1 Třídy: Task, TrainingSet a TrainingPattern

Úkolem knihovny je najít jeden nebo více neuronových stromů pro řešení konkrétního problému. Zadání tohoto úkolu je řešeno třídou *Task*. Tato třída obsahuje několik parametrů, z nichž pouze dva jsou pro uživatele povinné. Vynucení této povinosti je dáno konstruktorem třídy, kterému se tyto parametry předají jako argumenty. Pro použití knihovny je tedy nutné nejdříve vytvořit instanci této třídy. Tímto totiž knihovně řekneme, co po ní požadujeme.

Prvním povinným parametrem je *training_Set*. Ten představuje trénovací množinu, jejíž funkce a smysl byly popsány dříve. Trénovací množina je zastoupena třídou *TrainingSet*. Ta má dva konstruktory. První konstruktor umožňuje vytvořit prázdnou trénovací množinu. Jelikož každý trénovací vzor množiny obsahuje sadu vstupních hodnot a jím odpovídající sadu výstupních hodnot, musíme tomuto konstruktoru předat informaci o počtu vstupů a výstupů, se kterými budeme pracovat, a celkový počet trénovacích vzorů, které bude množina obsahovat. Počet výstupů udává také počet neuronových stromů, které má knihovna vytvořit. Jelikož konstruktor vytvořil prázdnou množinu, je potřeba do ní ještě přidat jednotlivé trénovací vzory. Pro vytváření trénovacích vzorů slouží třída *TrainingPattern*. Té v konstruktoru předáme počet vstupů a výstupů neuronové sítě a poté nastavíme vstupní a výstupní hodnoty. Trénovací vzory můžeme také vytvořit jako dvourozměrné pole typu „double“. Potom využijeme druhý konstruktor třídy *TrainingSet*, kterému předáme jako argument toto pole. Dále můžeme pojmenovat jednotlivé vstupy a výstupy trénovacích vzorů. K tomu slouží veřejné vlastností *InputNames* a *OutputNames*.

Druhým povinným parametrem třídy *Task* je parametr *maxError*. Ten udává maximální povolenou chybu, které se neuronový strom při učení může dopustit. Velikost maximální povolené chyby je tudíž hlavní podmínkou, která říká, kdy se může proces vývoje neuronového stromu zastavit.

Ostatní parametry jsou nepovinné a jsou přednastavené na hodnoty, které byly stanoveny dle praktické zkušenosti získané při vývoji knihovny. Knihovnu tedy může použít jak zkušený uživatel, který si může parametry nastavit podle svých vlastních potřeb, tak i nezkušený uživatel, který se nemusí v dané oblasti příliš orientovat a přesto může dosáhnout uspokojivých výsledků. Význam jednotlivých parametrů bude popsán v dalších sekcích týkajících se tříd, které tyto parametry používají. Zde ještě zmíním parametry *lock* a *abort*. Parametr *Lock* slouží k uzamčení instance třídy *Task*. Instance se uzamkne automaticky, když se spustí proces hledání řešení, ale může být zamčena i uživatelem

pomocí metody *lockTask*. Po uzamčení se nedají nastavovat parametry. Je to proto, aby se neměnily důležité hodnoty v průběhu procesu. Zabrání se tak nekonzistenci a nepředvídanému chování procesu. Jediný parametr, který se dá po uzamčení měnit, je parametr *abort*. Ten slouží pro předčasné ukončení procesu nezávisle na tom, jestli se našlo nebo nenašlo vhodné řešení.

4.5.2 Třídy: Fnt, Result, ProcessStatus a ResultStatus

Třída *Fnt* je vstupním bodem knihovny. Její hlavní metodou je metoda *findSolutionForTask()*, která spouští proces hledání vhodného řešení pro zadaný úkol. Úkol neboli zadání (instance třídy *Task*) se předává instancí třídy *Fnt* při vytváření v konstruktoru. Po zavolání metody *findSolutionForTask()* se uzamkne instance třídy *Task* a spustí se vlastní proces. V průběhu procesu generuje instance třídy *Fnt* dva typy událostí.

Událost *ProcessChanged* je vyvolána pokaždé, když se změní stav procesu. Stavy, kterými může proces procházet, jsou dány výčtem *ProcessStatus*. Když se proces spustí, je vyvolána událost se stavem *RUNNING*. Pokud proces doběhne do konce, je vyvolána událost se stavem *FINISHED*. Je-li proces přerušen dříve, pomocí příznaku *abort*, pak je vyvolána událost se stavem *ABORTED*. Třída, nesoucí informaci o stavu procesu, se jmenuje *StatusData* a je součástí generované události.

Událost *EpochStarted* je vyvolána pokaždé, když začne nová epocha vývoje neuronového stromu. Pojem epocha bude upřesněn později. Tato událost obsahuje informace o aktuálním „dění“, které jsou součástí třídy *InfoData*. Odsud se dozvíme aktuálně dosaženou chybu nejlepšího neuronového stromu, která epocha právě běží, kolikátý neuronový strom se aktuálně vyvíjí (v případě, že máme více výstupů) a celkový počet dosud vygenerovaných neuronových stromů. Tato událost také nepřímo říká, že aplikace pracuje.

Uvedené události lze využít nejen k informativním účelům, ale také k řízení aplikace, která knihovnu implementuje. Na základě aktuálních výsledků a stavu procesu se může externí aplikace rozhodovat jaký, bude další její postup.

Jakmile proces skončí, a je jedno jestli doběhne sám nebo je zastaven příznakem *abort*, vytvoří instanci třídy *Result*, do které uloží jeden nebo více výsledků řešení a doplňující informace. Pod pojmem „řešení“ budeme nyní myslet jeden nebo více neuronových stromů, které jsou od knihovny požadovány. Důležitou informací v instanci třídy *Result* je vlastnost *Status*, která říká, jak dopadl výsledek procesu. Status výsledku je dán jednou ze dvou hodnot výčtu *ResultStatus*. Pokud se nalezlo řešení, které splňuje požadavek na maximální povolenou chybu, obsahuje stav výsledku hodnotu *FOUND*. Pokud má nejlepší řešení chybu větší než maximální povolenou, obsahuje stav výsledku hodnotu *NOT_FOUND*. V případě stavu *FOUND* obsahuje vlastnost *AppropriateTrees* jedno nebo více řešení. V případě stavu *NOT_FOUND* obsahuje pouze jedno řešení. Je to doposud nejlepší nalezené řešení, i když se nepodařilo dosáhnout požadované chyby. Výsledkem procesu hledání je tedy vždy minimálně jedno řešení.

Pokud výsledek procesu obsahuje více než jedno řešení, je možné je seřadit. Na výběr je řazení podle velikosti chyby metodou *sortByError()* nebo podle velikosti neuronového stromu (počtu použitých neuronů) pomocí metody *sortBySize()*.

Dále třída *Fnt* obsahuje metodu *saveNeuralTreesToFile()*, která uloží výsledné řešení do XML souboru. Kromě řešení v podobě pole neuronových stromů musíme metodě také předat název souboru a adresáře, kam se má řešení uložit. Poslední důležitou metodou třídy *Fnt* je metoda *loadNeuralTreesFromFile()*, která načte uložené řešení ze souboru a vytvoří z něj potřebné objekty. Tato metoda má pouze jeden argument a tím je cesta k souboru.

4.5.3 Třídy: Neuron, Connection a NeuralTree

Tyto tři třídy představuji model neuronového stromu.

Třída *Neuron* představuje formální neuron. Každý neuron má svoje identifikační číslo zastoupené vlastností *Id*, které je unikátní v rámci jednoho neuronového stromu. Toto číslo je předáno konstruktoru při vytváření objektu neuronu. Číslo je důležité a musí být vždy nastaveno na určitou hodnotu. Čísla se neuronům přiřazují od jedničky nahoru při vytváření neuronového stromu. Začíná se u vstupních neuronů a pokračuje se přes neurony ve skrytých vrstvách až nakonec k neuronu výstupnímu. Vstupní neurony tak mají nejnižší čísla a pomoci těchto čísel (identifikátorů) knihovna rozlišuje, o který vstup se jedná. To je také směrodatné pro uživatele. Ten sice s objektem neuronu nepřijde přímo do styku, ale bude zadávat vstupní vektor hodnot celému stromu, aby získal jeho odpověď. Při předávání tohoto vektoru vstupů musí dodržet pořadí jednotlivých hodnot. Toto pořadí je dáno právě identifikačními čísly neuronů a je vždy vzestupné. Důležitost identifikačních čísel se projevuje hlavně při načítáni neuronového stromu z XML souboru, kde mohou být neurony umístěny v náhodném pořadí. Nejvyšší identifikační číslo má výstupní neuron. Jeho číslo je použito pro výpočet hašovací funkce. Identifikační čísla neuronů jsou také použity při ukládání neuronového stromu do souboru XML. Dále neuron obsahuje parametry jako je stav, práh, potenciál, strmost, aktivační funkce a také spoje k ostatním neuronům. Aktivační funkce je dána výčtem *ActivationFunction* a může nabývat hodnot *RADIAL_BASIS_FUNCTION* (funkce radiální báze), *STANDARD_SIGMOID_FUNCTION* (funkce standardní sigmoida) a *HYPERBOLIC_TANGENT_FUNCTION* (funkce hyperbolický tangens). Práh neuronu odpovídá parametru „a“ v rovnici 10 a strmost parametru „b“.

Třída *Connection* představuje spojení dvou neuronů. Obsahuje parametr *inputNeuron*, který odkazuje na vstupní neuron. „Vstupní“ ve vztahu k danému spoji. Dále třída *Connection* obsahuje parametr *weight*, představující váhu spoje.

Třída *NeuralTree* představuje jeden neuronový strom, který se skládá z neuronů a spojů. Nejdůležitější metodou této třídy je metoda *getResponse(double[] inputValues)*, která vrací

odezvu neuronového stromu na vstupní podnět. Metodě předáme jako argument pole typu *double*, které představuje vstupní vektor. Neuronový strom (v této knihovně) pracuje na vstupu s reálnými hodnotami v intervalu $\langle -1; 1 \rangle$. Výsledkem volání metody *getResponse* je opět hodnota typu *double* z intervalu $\langle -1; 1 \rangle$ nebo případně $\langle 0; 1 \rangle$ v závislosti na použité aktivační funkci neuronu. Tato hodnota je výstupem neuronového stromu. Vlastnost *Title* umožňuje pojmenovat neuronový strom. Titulek stromu tak může sloužit k jeho odlišení od ostatních. V aplikaci samotné se to asi nevyužije, ale v XML souboru, kde bude uloženo řešení, může být neuronových stromů více a pak by nemuselo být jasné, který strom je který.

Konstruktor třídy *NeuralTree* je pouze interní, protože uživatel sám neuronový strom nevytváří. K dispozici je metoda *equals(NeuralTree anotherTree, bool onlyStructure)*, která porovnává dva neuronové stromy. Volá se na neuronovém stromu a její první argument je odkaz na druhý neuronový strom se kterým se bude porovnávat. Druhý argument říká, jestli se budou porovnávat pouze struktury stromů nebo i parametry. Metoda *ToString()* slouží k převedení struktury stromu a jeho parametrů do podoby řetězce. Ten pak může být zobrazen třeba na konzoli. Je to jediná funkce knihovny, která umožňuje zobrazit neuronový strom. Toto zobrazení není moc přehledné, ale jako základní informace o tom, jak daný strom vypadá, to postačuje. Za zmínku stojí vlastnost *Size*, která vrací počet „použitých“ neuronů ve stromu. Slovo použitých je podstatné, protože některé ze vstupních neuronů nemusí být využity. Nemají spoje k neuronům do skrytých vrstev. Velikost neuronového stromu je o tyto nepoužité neurony menší. Seznam ID všech použitých vstupních neuronů obsahuje vlastnost *UsedInputs*.

Vlastnosti *InputNeurons* a *OutputNeuron* nejsou pro uživatele v základním režimu potřeba. Jsou využívány především interně knihovnou. Slouží pro přístup ke vstupním neuronům a k výstupnímu neuronu. Tyto dvě vlastnosti může uživatel využít v případě, že problém, který hodlá řešit pomoci neuronové sítě, je příliš složitý a lze jej rozdělit do několika menších podproblémů. Pro každý podproblém se nalezne jeden neuronový strom a tyto stromy se pak mezi sebou propojí. Doba pro nalezení jednodušších neuronových stromů je totiž podstatně kratší. Propojení dvou neuronových stromů se dá realizovat tak, že se vytvoří nový spoj (instance třídy *connection*) a nastaví se mu jako vstupní neuron výstupní neuron jednoho stromu, který se získá právě díky vlastnosti *OutputNeuron*. Poté se nový spoj přidá k některému ze vstupních neuronů druhého stromu. Ten získáme zase pomocí vlastnosti *InputNeurons*.

Princip zpracování podnětů
Po zavolání metody *getResponse* se vstupní vektor předá vstupním neuronům. Hodnoty se neuronům předají pomoci jejich vlastnosti *State*, takže se vlastně přímo nastaví jejich stav. Vstupní neurony neprovádějí žádnou jinou funkci než předání vstupních hodnot dalším neuronům. Následně se na výstupním neuronu zavolá metoda *getState()*. Ta volá rekurzívně stejnou metodu ve všech svých „potomcích“. Při počítání výstupní hodnoty neuronového stromu je tak použit algoritmus průchodu stromem do hloubky. Jelikož

je pro reprezentaci formálního neuronu zavedena pouze jedna třída, nelze přímo rozpoznat, který neuron je vstupní a který ne. Tato vlastnost se rozpozná za běhu programu tak, že vstupní neurony nemají žádné spoje, krom jednoho, který je na jejich výstupu. Díky této dynamice je možno provádět spojování neuronových stromů do větších struktur, jak bylo popsáno dříve. Proto musí metoda *getState()* nejdříve ověřit, jestli má daný neuron nějaké spoje. Pokud ano, znamená to, že se nejdříve musí získat výstupní stavy neuronů v nižší vrstvě. Proto se v cyklu volá přes všechny spoje metoda *gatState* na neuronech v nižší vrstvě. Získané hodnoty se vynásobí váhou spoje a sečtou se. Pak se od nich odečte hodnota prahu a výsledek se uloží jako potenciál neuronu. Jakmile je znám potenciál neuronu, zavolá se metoda *transfer()*, která z potenciálu vypočte stav svého neuronu pomoci aktivační funkce. Konkrétní aktivační funkce se neuronu nastavuje automaticky při vytváření neuronového stromu podle informace v objektu *Task*.

Konverze vstupních a výstupních hodnot
Jak již bylo uvedeno, neuronový strom je navržen pro práci s reálnými hodnotami. Ty však mohou být pouze v intervalu $\langle -1; 1 \rangle$. Pokud je potřeba pracovat s jiným intervalem hodnot, musí se provést konverze celého intervalu na interval $\langle -1; 1 \rangle$. Na toto omezení se musí brát ohled už při vytváření trénovaci množiny. Výstup neuronového stromu je v tomto intervalu pouze za předpokladu, že se pro neurony použije jako aktivační funkce Hyperbolický tangens. Pokud se použije funkce Standardní sigmoida nebo Radiální báze, pak je interval výstupních hodnot $\langle 0; 1 \rangle$.

4.5.4 Parametr sizeOfBunch

Než budu pokračovat dále, je třeba objasnit význam parametru *sizeOfBunch*, který se nachází ve třídě *Task*. Tento parametr významně ovlivňuje chování knihovny a pokládám tedy za důležité jej dobře objasnit.

Jelikož máme vytvořit neuronový strom, který se musí naučit příklady z trénovací množiny, o které nevíme, co vlastně představuje, nastává problém, že nevíme kolik neuronu k tomu budeme potřebovat. To je sice předmětem optimalizačního algoritmu, ale ten je založen na principech genetiky, která používá DNA řetězec pro zakódování struktury neuronového stromu. Abychom mohli používat operátor křížení, musí mít délku DNA řetězce každý strom stejnou. Na začátku tedy musíme zvolit určitou délku DNA řetězce, se kterou se bude po celou dobu procesu pracovat. Délka DNA řetězce zároveň omezuje maximální počet neuronů, které jdou v takovém řetězci zakódovat. Abychom nezvolili příliš malou délku řetězce (neurony by se do ní nevešly), ale ani příliš velkou, která by zbytečně zvyšovala nároky na výpočetní prostředky a prodlužovala výpočet samotný, potřebujeme alespoň přibližnou představu o tom, kolik neuronů budeme potřebovat.

Můžeme odhadnout počet potřebných neuronů na základě praxe, ale přesnější bude, když nám uživatel sám řekne, jak moc je jeho problém složitý. Protože tato knihovna počítá jak s méně zkušeným uživatelem, tak i se zkušeným uživatelem, nabízí oba přístupy k přibližnému odhadu počtu neuronů. V případě, že knihovnu použije zkušený uživatel, má

možnost „říct“ knihovně něco o složitosti problému. Na tomto základě tak knihovna odhaduje maximální počet neuronů. K tomu právě slouží již zmiňovaný parametr *sizeOfBunch*. V překladu to znamená *velikost svazku*. Velikost svazku udává počet trénovacích vzorů na jeden svazek. Když stanovíme, kolik vzorů bude obsahovat jeden svazek, můžeme tak celou trénovací množinu rozdělit do několika svazků. Celkový počet svazků trénovací množiny chápe knihovna jako číslo vyjadřující míru složitosti problému. Čím je toto číslo větší, tím je problém složitější. Mezi počtem všech svazků množiny a velikostí jednoho svazku je nepřímá úměra. Takže lze říci, že čím menší počet vzorů (pravidel) je v jednom svazku, tím více svazků pak množina obsahuje. Význam parametru *sizeOfBunch* je tedy říci knihovně míru složitosti problému. Jeho hodnota tak zásadně ovlivňuje délku procesu, jeho náročnost na výpočetní prostředky a jeho celkový výsledek (upřesním později). Tato myšlenka vychází z následujícího příkladu.

Máme dvě trénovací množiny. Každá obsahuje stejný počet trénovacích vzorů. Jak již bylo řečeno v kapitole 2.9, *Využití neuronových sítí*, jednotlivé vzory trénovací množiny představují body v prostoru. Tyto body se snaží neuronová síť proložit hyperplochou při procesu učení. Pokud jsou body v prostoru rozloženy tak, že hyperplocha, která je má protínat, není složitá, postačí nám méně neuronů pro tuto realizaci. V případě, že body v prostoru potřebují složitější hyperplochu (funkci) k proložení, potřebujeme neuronů více. Proto tyto dvě trénovací množiny se stejným počtem trénovacích vzorů nemusí představovat stejně složité problémy. Algoritmus knihovny dostane na vstupu trénovací množinu, z níž má pouze tři přímé informace. Počet vstupů budoucí neuronové sítě (neuronového stromu), počet výstupů budoucí sítě (počet jednotlivých stromů) a počet trénovacích vzorů. Dá se předpokládat, že čím více trénovacích vzorů bude obsahovat trénovací množina, tím složitější bude hyperplocha k proložení těchto vzorů. Rovněž se dá předpokládat, že i více vstupů a výstupů bude znamenat složitější problém. Mohli bychom sice prozkoumat rozložení bodů (trénovacích vzorů) v prostoru a podle toho posoudit složitost problému, ale toto může být složité. Z tohoto důvodu může algoritmus odhadnout složitost problému na základě tří předchozích údajů. Protože však uživatel knihovny vytváří trénovací množinu a tudíž by mohl mít představu o její složitosti, může knihovně tuto informaci poskytnout.

Je tedy na uvážení uživatele, aby zadal určitou hodnotu parametru *sizeOfBunch*. Jak tuto hodnotu zvolit je otázka praxe. Pokud jí uživatel nezadá, použije algoritmus hodnotu přednastavenou. Nicméně je-li uživatel v této oblasti nezkušený, může ovlivnit maximální počet neuronů ve stromu parametrem *multiplicator*. Jak tyto parametry ovlivňují činnost algoritmu bude popsáno později.

4.5.5 Třídy: Breeder, Teacher, Evolution a FntFactory

Metoda *findSolutionForTask()* třídy *Fnt* spouští hledání vhodného řešení. Tuto činnost však sama nevykonává, ale pověří tím instanci třídy *Breeder* (šlechtitel). Ve třídě *Breeder* se nachází hlavní smyčka procesu. Ta se spouští v samostatném vlákně a skládá se ze čtyř vnořených cyklů. První cyklus opakuje proces hledání vhodného neuronového stromu

pro každý požadovaný výstup. Počet těchto opakování je odvozen od počtu výstupních hodnot v trénovací množině. Druhý cyklus je vnořen do prvního a nazývá se *Epocha*. Epocha v sobě obsahuje zbylé dva cykly. Jeden se nazývá *Generace* a implementuje optimalizaci struktury neuronového stromu. Druhý opakuje proces učení pro každý neuronový strom v populaci. Vývoj neuronového stromu probíhá v instanci třídy *Evolution*. Zde je implementován evoluční algoritmus, používající genetické operátory, pro optimalizaci struktury neuronového stromu. Třída *Evolution* udržuje populaci neuronových stromů v počtu, který je dán parametrem *populationSize*. Tento parametr je ve třídě *Task* a jeho nejmenší hodnota může být osm. To dovoluje vytvořit minimální počet nových jedinců v populaci o počtu dva. Algoritmus má totiž nastaven maximální počet nových jedinců v jednom kroku ve velikosti čtvrtiny populace. Aby tedy mohli vzniknout alespoň 2 noví jedinci v jednom kroku, je nutné mít nejnižší populaci v počtu osm.

Než započne cyklus hledání vhodného řešení, vygeneruje se počáteční populace neuronových stromů, skládající se z náhodných struktur a parametrů. K vytváření neuronových stromů slouží třída *FntFactory*. Její metoda *buildRandomTree()* „vystaví“ náhodný neuronový strom. Tato metoda se používá pouze při generování počáteční populace. Pro vytváření stromů na základě DNA se používá metoda *buildNeuralTreeFromDna()*.

Generování náhodného stromu
K vygenerování prvotní populace náhodných jedinců, jak bylo řečeno, použijeme třídu *FntFactory*. Než začneme vytvářet jedince (neuronové stromy), je potřeba stanovit délku řetězce DNA, který bude tyto jedince kódovat. Tuto délku lze stanovit na základě maximálního počtu neuronů, které může strom obsahovat. Maximální počet neuronů ve stromu zjistíme sečtením vstupních neuronů, jednoho výstupního neuronu a maximálního počtu neuronů ve skrytých vrstvách. Matematicky vyjádřeno:

$$N_{all} = N_{in} + N_{out} + N_{hidden} \tag{14}$$

Kde:
N_{all} - je maximální počet všech neuronů ve stromu
N_{in} - počet vstupních neuronů
N_{out} - počet výstupních neuronů (v tomto případě pouze jeden)
N_{hidden} - maximální počet neuronů ve skrytých vrstvách

Výstupní neuron je jen jeden a počet vstupních neuronů je dán trénovací množinou. Zbývá tedy zjistit maximální počet skrytých neuronů. Tento počet odhadneme tak že vynásobíme počet vstupních neuronů parametrem ***multiplicator*** a počtem svazků. Multiplikátor neboli násobitel se volí náhodně podle praktického uvážení.

$$N_{hidden} = N_{in} * m * B \tag{15}$$

Kde:
N_{hidden} - maximální počet neuronů ve skrytých vrstvách
N_{in} - počet vstupních neuronů

m - multiplicator (násobitel)
B - počet svazků

Maximální počet skrytých neuronů vychází z počtu vstupních neuronů. Ten je daný pevně a nemůžeme jej změnit. Můžeme ale měnit násobitel a nepřímo také počet svazků. Počet svazků vypočteme z trénovací množiny na základě velikosti svazku. Ten byl popsán již dříve.

$$B = (P_{all}/P_B) + 1 \tag{16}$$

Kde:
B - počet svazků
P_{all} - celkový počet trénovacích vzorů
P_B - počet trénovacích vzorů v jednom svazku (velikost svazku)

Počet svazků B se po výpočtu zaokrouhlí na nejblížší celé číslo směrem dolů. Aby se nestalo, že po zaokrouhlení bude počet svazků 0, přičítá se k tomuto počtu jednička. Jelikož se multiplikátor a velikost svazku nastavuje víceméně náhodně, je možné, že se v průběhu procesu hledání nenajde řešení s požadovanou chybou, protože jsme povolili příliš malý počet neuronů, které se mohou ve stromu vyskytovat. V takovém případě můžeme proces opakovat s jinou (vyšší) hodnotou násobitele. Také můžeme nastavit nižší velikost svazku. Vhodnější se však může jevit nastavení násobitele na určitou hodnotu a tu pak neměnit. Místo toho raději měnit hodnotu velikosti svazku a hledat tak optimální velikost svazku. Je to z toho důvodu, že když se později rozhodneme přidat trénovací vzory do trénovací množiny, tak se hodnota počtu svazku automaticky zvětší a my nemusíme pamatovat na to, že je třeba zvýšit maximální počet neuronů. Toto může nastat například, když nebudeme spokojeni s výsledným řešením z důvodu nevhodně zvolené trénovací množiny. Rozhodneme se tedy množinu upravit a proces spustit znovu se stejným nastavením parametrů.

Velikost svazku má vliv i na minimální počet neuronů ve stromu. Zatím co zvyšování hodnoty násobitele zvětšuje pouze maximální počet neuronů ve stromu, tak snižování velikosti svazku nejen že zvyšuje maximální počet neuronů ve stromu, ale zvyšuje také minimální počet neuronů ve stromu. Toto chování by mohlo urychlit proces tím, že se neztrácí čas s „velmi malými“ strukturami, které nejsou schopny se naučit trénovacím vzorům. Konkrétně se minimální počet neuronů ve skrytých vrstvách stanoví jako počet svazků.

Řetězec DNA v tomto případě představuje pole čísel typu *integer* čili pole celých čísel. Struktura neuronového stromu je do tohoto pole zakódovaná jako „rozložená“ matice spojů mezi neurony. Sloupce matice spojů představují všechny potenciální neurony ve stromu seřazené vzestupně. Počet sloupců je tedy roven maximálnímu počtu všech neuronů ve stromu. Řádky matice spojů pak představují skryté neurony a výstupní neuron, seřazeny taktéž vzestupně. Řádky odpovídající vstupním neuronům nejsou potřeba, jelikož vstupní neurony nemají žádné potomky a tudíž do nich nemůže vést žádný vstupní

spoj. Hodnoty matice pak mohou nabývat čísel 0 nebo 1. Jednička v dané buňce znamená, že od neuronu v daném sloupci (neuron nižší vrstvy) vede spoj k neuronu v daném řádku (neuron vyšší vrstvy). Nula v dané buňce pak znamená, že mezi danými neurony spoj není. Délku DNA řetězce tedy vypočteme podle následujícího vzorce.

$$l_{DNA} = N_{all} * (N_{hidden} + N_{out}) \tag{17}$$

Kde:
l_{DNA} - je délka řetezce DNA (počet prvků pole)
N_{all} - je maximální počet všech neuronů ve stromu
N_{hidden} - maximální počet neuronů ve skrytých vrstvách
N_{out} - počet výstupních neuronů (v tomto případě pouze jeden)

Matici spojů vložíme do pole DNA tak, že postupně bereme její řádky shora a vkládáme je do pole zleva.

Nyní můžeme začít vytvářet náhodné struktury neuronových stromů (jedince). Algoritmus pro tuto činnost funguje takto: Vygeneruje se náhodné číslo v rozsahu (B, N_{hidden}), které představuje aktuální počet skrytých neuronů v konkrétním stromu. Vytvoří se všechny vstupní neurony. Pak se vygeneruje opět náhodné číslo v rozsahu $(1, N_{unused})$, které představuje počet skrytých neuronů v aktuálně budované vrstvě. N_{unused} zde znamená počet dosud nepoužitých neuronů. Na začátku se rovná počtu všech neuronů ve skrytých vrstvách. Pak se v průběhu vytváření stromu (po vytvoření každé skryté vrstvy) snižuje o počet neuronů, které již byly použity v předchozích vrstvách. Vytvoří se tedy všechny neurony budované vrstvy a náhodně se propojí s neurony v nižší vrstvě. Takto se postupuje vrstva po vrstvě až k výstupnímu neuronu. Mezitím se ještě s jistou pravděpodobností propojují vstupní neurony s neurony ve vyšších vrstvách (spojení přes vrstvy) a dokonce i přímo s neuronem výstupním. Pravděpodobnost propojení vstupních neuronů do skryté vyšší vrstvy je možno ovlvnit parametrem *inputToHiddenProbability*. Pravděpodobnost propojení vstupních neuronů přímo s výstupním neuronem je možno ovlivnit parametrem *inputToOutputProbability*. Oba parametry se nastavují ve třídě *Task*. Počet vrstev a počet neuronů v jednotlivých vrstvách je náhodný. Prahy a strmosti všech neuronů a váhy všech spojů jsou nastaveny náhodně v intervalu $\langle 0; 1 \rangle$. Aktivační funkce je dána parametrem *function* a to všem neuronům stejně. Abych to upřesnil, tak ve vzorci 10 odpovídá parametr *a* prahu neuronu a parametr *b* odpovídá strmosti neuronu.

Generování stromu z DNA
Když předchozí metoda *buildRandomTree()* vygeneruje náhodný neuronový strom, zapíše do něj také jeho DNA řetězec. Tento řetězec obsahuje pouze informaci o struktuře stromu, která je dána počtem neuronů a použitými spoji. Po vygenerování celé počáteční populace těchto stromů se již náhodné stromy nevytvářejí. Místo toho se nová populace získává křížením již existujících jedinců. To probíhá tak, že se vezmou DNA řetězce dvou náhodně vybraných jedinců a vytvoří se z nich dva nové řetězce. Tyto nové řetězce mají každý určitou část DNA od jednoho i druhého ze svých „rodičů“ (původních řetězců). Z

těchto řetězců se pak vytvoří dva nové neuronové stromy. To zajišťuje metoda *buildNeuralTreeFromDna(int[] dna)*. Té se předá řetězec DNA jako argument. Jelikož je nový řetězec DNA předaný metodě složen ze dvou částí různých jiných řetězců, nemusí některé jeho částí dávat smysl při vytváření nového jedince. Záleží na tom, ve kterých úsecích se oba původní řetězce rozdělily. Navíc na řetězce DNA působí i mutace, která může mít i negativní vliv na výsledného jedince. Z těchto důvodů má metoda pro vytváření stromu z DNA řetězce k dispozici opravné mechanismy. Musí se zajistit, aby každý neuron v nově vytvořeném stromu, který je listem tohoto stromu, ale není to vstupní neuron, byl odstraněn. Takovýto neuron totiž neplní žádnou funkci a narušoval by správnou činnost neuronového stromu při vytváření odezvy. Dále se musí ze stromu odstranit všechny „zdvojené" cesty k určitému neuronu ve skryté vrstvě a cykly. Cykly by způsobily nekonečnou smyčku čtení výstupní hodnoty při odezvě a protože je proces rekurzivní, došlo by k vyčerpání přidělené paměti. Při odstraňování zdvojených cest a cyklů je většinou na výběr více možností, kterou cestu ponechat a kterou odstranit. Výběr se ponechává náhodě, proto pokud by se z jednoho DNA řetězce mělo vytvořit více jedinců, mohli by mít jedinci navzájem různé struktury. Po vygenerování struktury neuronového stromu z DNA řetězce se nastaví hodnoty prahů a strmosti jednotlivých neuronů a hodnoty vah všech spojů na náhodná čísla.

Evoluční procesy
Po vygenerování náhodné prvotní populace se v cyklech zvaných *Epocha* provádějí následující kroky. Nejdříve se spustí cyklus optimalizace struktury neuronových stromů zvaný *Generace*. V tomto cyklu se ohodnotí všichni jedinci v populaci podle zadané funkce zdatnosti. Funkce zdatnosti, neboli *fitness function*, se volí ve třídě *Task* parametrem *fitness*. Funkce zdatnosti je daná výčtem *ActivationFunction* a může nabývat hodnot *RADIAL_BASIS_FUNCTION*, *STANDARD_SIGMOID_FUNCTION* a *HYPERBOLIC_TANGENT_FUNCTION*. Po ohodnocení probíhá výběr jedinců ke křížení. Výběr jedinců je náhodný proces, při kterém mají větší pravděpodobnost výběru jedinci s lepší hodnotou funkce zdatnosti. V tomto případě je hodnota funkce zdatnosti vypočítaná z průměrné nebo maximální chyby odezvy neuronového stromu. Znamená to tedy, že čím menší chybu má daný jedinec, tím větší je jeho zdatnost. Když jsou jedinci vybráni, proběhne proces křížení na základě jejich DNA řetězců. Výsledkem jsou nové řetězce DNA. Na nové DNA řetězce se poté s jistou pravděpodobnosti aplikuje operátor mutace. Ten probíhá tak, že se vybere náhodné místo v řetězci (gen) a invertuje se jeho hodnota. Pravděpodobnost mutace je dána pevně (5%), nelze ji tedy měnit. Na základě nově připravených řetězců DNA se vytvoří noví jedinci, kteří se zařadí do existujíci populace. Jelikož musí být v populaci neustále stejný počet jedinců musí se odstranit stejný počet starých jedinců. Jedinci, kteří se z populace musí odstranit, se opět vyberou na základě náhodného výběru. Tentokrát mají větší pravděpodobnost výběru jedinci se slabší hodnotou zdatnosti. Tímto vznikne nová populace a generační cyklus se opakuje. Opět se provede ohodnocení jedinců atd. Počet těchto generačních cyklů v jedne epoše je dán parametrem *maxGenerations*. Při této činnosti se v populaci udržuje několik jedinců, kteří patří do podmnožiny zvané „elita". Počet těchto elitních jedinců je dán parametrem *eliteSize* a

nemůže být větší než polovina populace. Do elity patří ti nejsilnější jedinci. Tito jedinci se z populace nikdy neodstraňují, protože představují neuronové stromy s aktuálně nejlepší strukturou. Vyřadit z „elitní jednotky“ je ale může jiný (mladší) jedinec, který dosáhne lepší zdatnosti. Ten tak nahradí původního eltního jedince sebou samým. Toto chování zajistí, že nikdy nepřijdeme o ta nejlepší řešení doposud nalezená.

Poté, co skončí cyklus optimalizace struktury, ověří se, jestli se v populaci nachází alespoň jeden neuronový strom s chybou menší nebo rovnou maximální povolené chybě. Pokud ano ukončí se proces hledání a všecny neuronové stromy, které mají chybu menší nebo rovnu maximální povolené chybě se uloží do objektu *Result*. Pokud takový neuronový strom neexistuje, zahájí se cyklus optimalizace parametrů neboli učení. Ten zajišťuje třída *Teacher*, která se snaží naučit každý neuronový strom v populaci řešit zadaný problém daný trénovací množinou. Učící cyklus se opakuje, dokud se nedosáhne chyby odezvy menší nebo rovné maximální povolené chybě. Maximální povolenou chybu třída získá z parametru *maxError*. Aby se nemarnil čas, strávený učením stromu, který není schopný se danému problému naučit (nemá na to vhodnou strukturu), je navíc stanoven maximální počet učících cyklů. Ten je dán parametrem *maxCyclesCount*. Podmínky pro zastavení učení na konkrétním stromu jsou tedy dvě a ty zajistí, že se učení zastaví buď po dosažení požadovaného výsledku nebo po vyčerpání limitu. V jednom kroku učícího cyklu se postupně prochází všemi trénovacími vzory a každý vzor se předloží neuronovému stromu jako podnět. Získaná odezva na podnět se porovná s předpokládanou odezvou a vypočte se rozdíl. Tento rozdíl představuje chybu, které se neuronový strom dopustil. Tato chyba se pomocí parciální derivace aktivační funkce šíří zpět až k neuronům první skryté vrstvy. Během tohoto šíření se celková velikost chyby „rozpadá“ na menší částí, ze kterých se vypočítávají odchylky jednotlivých vah spojů, prahů a strmostí neuronů. Tyto odchylky pak slouží ke korekci vah, prahů a strmostí. Třída *Teacher* tak vlastně implementuje algoritmus back-propagation, popsaný v kapitole 2.6. Když se provádí korekce vah, prahů a strmostí na základě vypočtených odchylek, nepoužívají se přímo vypočtené hodnoty odchylek, ale pouze jejich určitý poměr. Tento poměr je ovlivněn dvěma parametry, které je nutno rovněž stanovit na základě praktické zkušenosti. Matematicky je to vyjádřeno takto:

$$\delta = \delta_{calc} * \alpha + \delta_{prev} * \mu \tag{18}$$

Kde:
δ - je výsledná odchylka dané váhy spoje, prahu nebo strmosti neuronu, která se použije pro korekci
δ_{calc} - je vypočtená odchylka pro danou váhu, práh nebo strmost (výstup z derivace)
α - je koeficient učení
δ_{prev} - je velikost odchylky prahu, váhy nebo strmosti z předchozího kroku
μ - je vliv změny z předchozího kroku

Parametry α a μ jsou ve třídě *Task* a jsou již přednastavené. Uživatel si je však může změnit podle svých potřeb. Parametr *learningRate* zastupuje *koeficient učení* α a může být nastaven v intervalu hodnot $\langle 0; 1 \rangle$. Parametr *previousDeltaRate* zastupuje *vliv předchozího*

kroku μ a rovněž může být nastaven v intervalu $\langle 0; 1 \rangle$. Učící algoritmus navíc rozpoznává elitní stromy a těm je koeficient učení snížen na pětinu toho, který je nastaven ve třídě *task*. Vliv předchozího kroku se u elitních stromů snižuje na polovinu. Tyto dvě odlišnosti snižují pravděpodobnost, že dojde k poškození nejlepšího řešení učením.

I přes tato opatření, která minimalizují „ztrátu" nejlepšího jedince učením, se nedá zcela zabránit zhoršení chyby elitních jedinců. Proto existuje ve třídě *Task* parametr *FixTheBest-Specimen* („upevnit" nejlepšího jedince), který zajistí, že aktuálně nejlepší jedinec bude vyjmut z učícího cyklu. Je-li tento parametr nastaven na hodnotu „true", zamezí se učení nejlepšího jedince a tak není možno učením zhoršit jeho zdatnost.Toto řešení sebou přináší další problém a to ten, že nejlepší jedinec by se teoreticky mohl učením ještě více zlepšit, ale je mu to znemožněno. Proto se vždy v posledním generačním cyklu optimalizace struktury naklonuje aktuálně nejlepší jedinec a jeho kopie se dostane do cyklu učení. Pokud má tento jedinec potenciál se učením zlepšit, zlepší se jeho klon a nahradí tak nejlepšího jedince. V procesu učení tak máme třetí podmínku, kdy se učení na daném jedinci zastaví. Ta nastane, jakmile právě učený jedinec dosáhne lepší zdatnosti než nejlepší jedinec.

Cyklus hledání lepší struktury a učící cyklus jsou součástí cyklu zvaný *Epocha*, jak již bylo řečeno, a ten zajišťuje, že se neustále střídá optimalizace struktury a následně optimlizace parametrů. Maximální počet epoch je dán parametrem *maxEpochs*.

4.6 Struktura XML souboru

Výsledné řešení v podobě jednoho nebo více neuronových stromů je možno uložit do souboru pro jeho uchování a pozdější použití. Pro zápis byl zvolen značkovací jazyk XML. Struktura souboru byla zvolena tak, aby byla nezávislá na aplikaci, která se souborem pracuje a je následující: Kořenový element se jmenuje *Neural_forest*. V překladu neuronový les. Název je odvozen od neuronových stromů. Tento element nemá žádný atribut a může obsahovat pouze jeden nebo více elementů *Neural_tree*. Element *Neural_tree* představuje neuronový strom. Má jeden atribut a tím je *title*. Je určen k jednoduchému popisku stromu kvůli přehlednosti. Element *Neural_tree* může obsahovat jeden nebo více elemntů *Neuron*, který představuje formální neuron. Ten má parametry *id* zastupující identifikátor, *threshold* zastupující práh, *slope* zastupující strmost a *function* zastupující aktivační funkci. Element *Neuron* může obsahovat nula nebo více elementů *Connection*, představující spojení mezi neurony. Element *Connection* neobsahuje žádný element. Má pouze dva atributy. Atribut *weight* zastupuje váhu spoje a atribut *input* obsahuje id vstupního neuronu. Pojem vstupní neuron je myšlen z pohledu spoje, čili neuron, který je na vstupním konci spoje.

Při ukládání neuronového stromu do souboru se postupuje tak, že se nejdříve uloží všechny vstupní neurony v pořadí podle jejich identifikačních čísel. Poté se prochází stromem do hloubky a uloží se nejdříve všechny neurony ve skrytých vrstvách a nakonec neuron výstupní. Toto pořadí však není nutné. Metoda pro načítání neuronového stromu ze souboru je postavena tak, že si poradí i s náhodným pořadím neuronu v sou-

boru. Jediné, co by mělo být dodrženo, je počet vstupních neuronů. Trénovací množina obsahuje určitý počet vstupů v trénovacích vzorech. Od tohoto počtu se odvodí počet vstupních neuronů a ty všechny se uloží do souboru. Je možné, že proces hledání vhodného řešení nalezne takový neuronový strom, který bude splňovat požadavky a navíc nebude některé vstupy potřebovat. Teoreticky by se tyto „nepotřebné“ vstupy nemusely ukládat. Problém by však nastal v případě načtení a používání takového stromu na stejné trénovací nebo testovací množině. Stromu by se předkládalo více vstupních hodnot než by strom obsahoval. Navíc by pak nebylo jasné, který vstup je který. Pokud by se někdo rozhodl odstranit ze souboru nepotřebné vstupní neurony, měl by mít tuto skutečnost na paměti.

Níže je přiložen obsah XML souboru s uloženým jedním neuronovým stromem.

```
<?xml version="1.0" encoding="utf-8"?>
<Neural_forest>
    <Neural_tree  title ="6x - 3y + z">
        <Neuron id="1" threshold="0,1140126954363 72" slope="1" function="
            RADIAL_BASIS_FUNCTION" />
        <Neuron id="2" threshold="0,0853109765263791" slope="1" function="
            RADIAL_BASIS_FUNCTION" />
        <Neuron id="3" threshold="0,0274466844403402" slope="1" function="
            RADIAL_BASIS_FUNCTION" />
        <Neuron id="4" threshold="1,38582225390629" slope="1" function="
            RADIAL_BASIS_FUNCTION">
            <Connection weight="0,512331575383276" input="1" />
            <Connection weight="-0,258136749409693" input="2" />
            <Connection weight="0,830537841339158" input="3" />
        </Neuron>
        <Neuron id="5" threshold="1,32267221767749" slope="1" function="
            RADIAL_BASIS_FUNCTION">
            <Connection weight="0,975486855094029" input="4" />
        </Neuron>
    </Neural_tree>
</Neural_forest>
```

Výpis 1: Ukázka XML souboru s jedním neuronovým stromem

5 Testy výkonností a ukázka použití

Testy výkonnosti byly prováděny na sestavě PC:
CPU - Intel Pentium 4 - 2,4 GHz
RAM - 2GB
OS - Microsoft Windows XP

5.1 Realizace funkce sinus na omezeném intervalu

V tomto testu se pokusíme pomocí neuronového stromu simulovat funkci sinus. Test bude probíhat na intervalu $\langle 0°; 360° \rangle$.

5.1.1 Příprava testovací třídy

x	$\cos x$
0	0
45	0,7071
90	1
135	0,7071
180	0
225	-0,7071
270	-1
315	-0,7071
360	0

Tabulka 1: Trénovací množina 1

x	$\cos x$
0	0
22	0,3746
45	0,7071
80	0,9848
90	1
100	-0,1736
135	0,7071
158	0,3746
180	0
202	-0,3746
225	-0,7071
260	-0,9848
270	-1
290	-0,9396
315	-0,7071
338	-0,3746
360	0

Tabulka 2: Trénovací množina 2

x	$\cos x$
10	0,1736
40	0,6427
60	0,8660
133	0,7313
180	0
250	-0,9396
275	-0,9961
330	-0,5
360	0

Tabulka 3: Testovací množina

Pro účely testování použijeme samostatný projekt s názvem *TestFntLibrarySinus*, ve kterém nám stačí pouze jedna třída. Testovací třída obsahuje tyto metody:

- launchConsole() - spustí konzoli, která umožní ovládat testovací program
- getTrainingPatterns() - vytvoří dvourozměrné pole s trénovacími vzory
- getTestingPatterns() - vytvoří dvourozměrné pole s testovacími vzory (jiné než trénovací)

- convertPaterns() - provede konverzi vzorů na interval $\langle -1; 1 \rangle$
- findSolution() - nastaví parametry úkolu, zaregistruje se jako posluchač událostí *EpochSatrted* a *ProcessChanged*, spustí proces hledání řešení
- processInfoEvent() - reaguje na událost *EpochSatrted* a vypíše na konzoli aktuální stav nejlepšího neuronového stromu
- processStatusEvent() - reaguje na událost *ProcessChanged* a jakmile proces skončí, zavolá metodu *writeReport()*
- writeReport() - vypíše informace o výsledku procesu
- testSolution() - otestuje nalezené řešení a vypíše hodnocení
- generateErrorFile() - otestuje nalezené řešení pomocí více hodnot a uloží výsledky do souboru CSV

5.1.2 Test č.1

TrainingSet	č.1
MaxAllowedError	0,1
SizeOfBunch	2
Multiplicator	10
InputToOutputProbability	0.9
LearningRate	0,25
PreviousDeltaRate	0,55
MaxCyclesCount	200
Function	HYPERBOLIC_TANGENT_FUNCTION
Fitness	ROOT_MEAN_SQUARED_ERROR
PopulationSize	500
EliteSize	50
MaxGenerations	100
MaxEpochs	100

Tabulka 4: Nastavení parametrů třídy *Task* - test č. 1

V tomto testu je použita trénovací množina č.1. Maximální povolená chyba byla zvolena na 0,1. Jelikož má neuronový strom jen jeden vstup, tak se počet vstupů N_{in} podle vzorce 15, který použijeme pro odhad maximálního počtu skrytých neuronů, nijak neprojeví. Můžeme to ovlivnit pouze velikosti svazku a násobitelem. Velikost svazku byla pro začátek zvolena na hodnotu 2 a násobitel na hodnotu 10. Pravděpodobnost spojení vstupu přímo s výstupem byla nastavena na vysokou hodnotu, protože v tomhle případě je potřeba jediný vstup co nejvíce zapojit do činnosti. Koeficient učení a vliv předchozí změny jsou nastaveny na základě zkušeností. Menší koeficient učení nebude „rozkmitávat“ nastavování vah a prahů a vyšší vliv předchozího kroku urychlí učení. Maximální

počet cyklů učení je docela nízký, není potřeba se příliš zatěžovat učením, pokud strom nemá vhodnou strukturu. Aktivační funkce byla zvolena na hyperbolický tangens, protože její obor hodnot $\langle -1; 1 \rangle$ přesně odpovídá oboru hodnot funkce sinus. Fitnes funkce byla zvolena jako odmocnina ze střední kvadratické chyby, aby mohla být následně v dalším testu porovnána z funkcí maximální chyby. Velikost populace je záměrně „velká“, aby zvýšila pravděpodobnost různorodosti struktur neuronových stromů už od začátku procesu. Počet elitních jedinců je desetina populace, což by mělo stačit na uchování aktuálně nejlepších řešení. Pokud by byl příliš velký, byl by proces zdlouhavý, jelikož by se měnila jen malá část populace. Je nastaveno maximálně 100 generačních cyklů a poté následuje učení. Počet cyklů učení a optimalizace struktury jsem volil přibližně stejný. Obě optimalizace totiž mají vliv jedna na druhou, tak at' se pravidelně střídají. Při volbě maximálního počtu epoch jsem vycházel z velikosti populace a počtů jednotlivých cyklů. Volil sem tak, abych poskytl dostatečnou dobu na hledání.

Výsledky procesu
Tento test trval 85 sekund. V desíti epochách bylo vytvořeno 64 688 neuronových stromů. Jako nejlepší řešení byl vybrán neuronový strom s pořadovým číslem 12 354, který je složen z jedenácti neuronů. Tento strom je uložen v souboru „sinus_function_1.fnt“.

Pro ověření funkčnosti byla nejdříve použita trénovací množina a poté testovací množina. Čísla v tabulkách 5 a 6 představují odchylky výstupních hodnot od požadovaných hod-

Nejmenší odchylka	0,01359
Průměrná odchylka	0,09226
Median odchylky	0,07388
Maximální odchylka	0,17199

Tabulka 5: Statistika odpovědí pro trénovací množinu - test č. 1

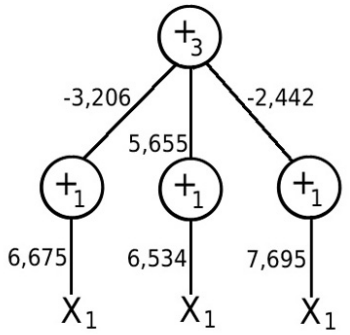
Nejmenší odchylka	0,02069
Průměrná odchylka	0,06673
Median odchylky	0,06171
Maximální odchylka	0,13582

Tabulka 6: Statistika odpovědí pro testovací množinu - test č. 1

not. V tabulce 5 je vidět, že maximální odchylka je větší než maximální povolená chyba, kterou jsme zadali v úkolu (třída *Task*). Je to tím, že jsme jako fitness funkci použili odmocninu ze střední kvadratické chyby. Celková chyba neuronového stromu se tak vypočítala jako průměrná chyba přes všechny trénovací vzory. Tudíž může být odpověď na některý vzor horší, než jsme požadovali. Z tabulky 6, kde jsou výsledky pro testovací množinu, tedy vzory, na které se neuronový strom neučil odpovídat, je vidět že v tomto případě jsou výsledky mírně lepší. Většinou to bývá naopak. Když se však podíváme na medián, zjistíme, že minimálně polovina odpovědí splňuje zadaný požadavek.

5.1.3 Test č.2

Požadujeme-li, aby neuronový strom odpovídal na všechny vzory z trénovací množiny s maximálně povolenou chybou danou v zadání, je nutné použít jako fitness funkci „maximální chyba“. Celková chyba neuronového stromu se pak vyhodnotí jako největší chyba,



Obrázek 16: Neuronový strom z testu č. 2

které bylo dosaženo průchedem přes vzory trénovací množiny.

V tomto testu tedy změníme pouze fitness funkci na hodnotu *MAX_ERROR* a spustíme proces znovu.

Výsledky procesu
Nyní test trval 316 sekund, prošel 33 epoch a bylo vytvořeno 209 198 stromů. Výsledný neuronový strom má 5 neuronů a je uložen v souboru s názvem „sinus_function_2.fnt". Na obrázku 16 je strom vidět i s hodnotami vah všech spojů. Vstupní neuron je označen symbolem x_1, ostatní neurony pak kroužky. Z tabulky 7 je nyní vidět, že i ta největší od-

Nejmenší odchylka	0,00099
Průměrná odchylka	0,04378
Median odchylky	0,04125
Maximální odchylka	0,09999

Tabulka 7: Statistika odpovědí pro trénovací množinu - test č. 2

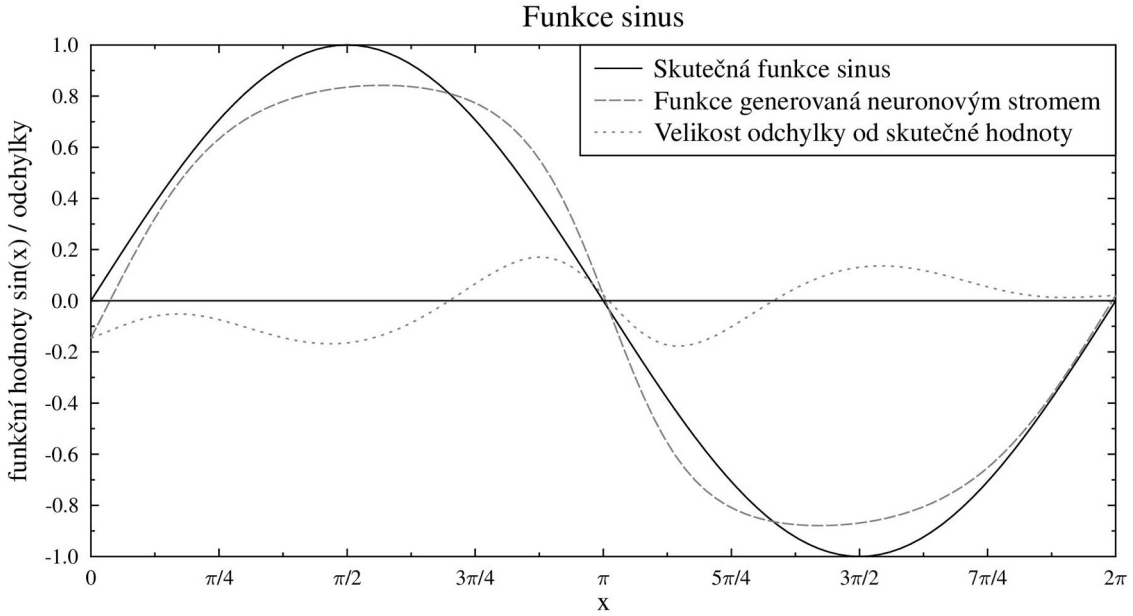
Nejmenší odchylka	0,00246
Průměrná odchylka	0,03306
Median odchylky	0,00928
Maximální odchylka	0,09687

Tabulka 8: Statistika odpovědí pro testovací množinu - test č. 2

chylka v odpovědi neuronového stromu splňuje náš požadavek na maximální povolenou chybu.

5.1.4 Test č.3

V tomto případě se pokusíme zlepšit odpovědi neuronového stromu na nenaučené vzory také tím, že přidáme vzory do trénovací množiny. Neuronová síťby v tomto případě měla lépe proložit křivku (hyperplochu) body roviny. Použijeme tedy trénovací množinu č.2. Dále se pokusíme zvýšit přesnonst v odpovědích snížením maximální povolené chyby na hodnotu 0,04. Protože teď budou nároky na přesnost vyšší, snížíme koeficient učení na hodnotu 0,15 a vliv předchozího kroku na hodnotu 0,45. Počet učících cyklů zvýšíme na hodnotu 300 a rovněž i maximální počet epoch na hodnotu 300.



Obrázek 17: Odpovědi neuronového stromu z testu č. 1

Výsledky procesu
Proces trval 58 minut. Prošel 166 epoch a vytvořil celkem 1 043 128 neuronových stromů. Výsledný strom má 16 neuronů a je uložen v souboru s názvem „sinus_function_3.fnt“.

Nejmenší odchylka	0,00132
Průměrná odchylka	0,02109
Median odchylky	0,01977
Maximální odchylka	0,03983

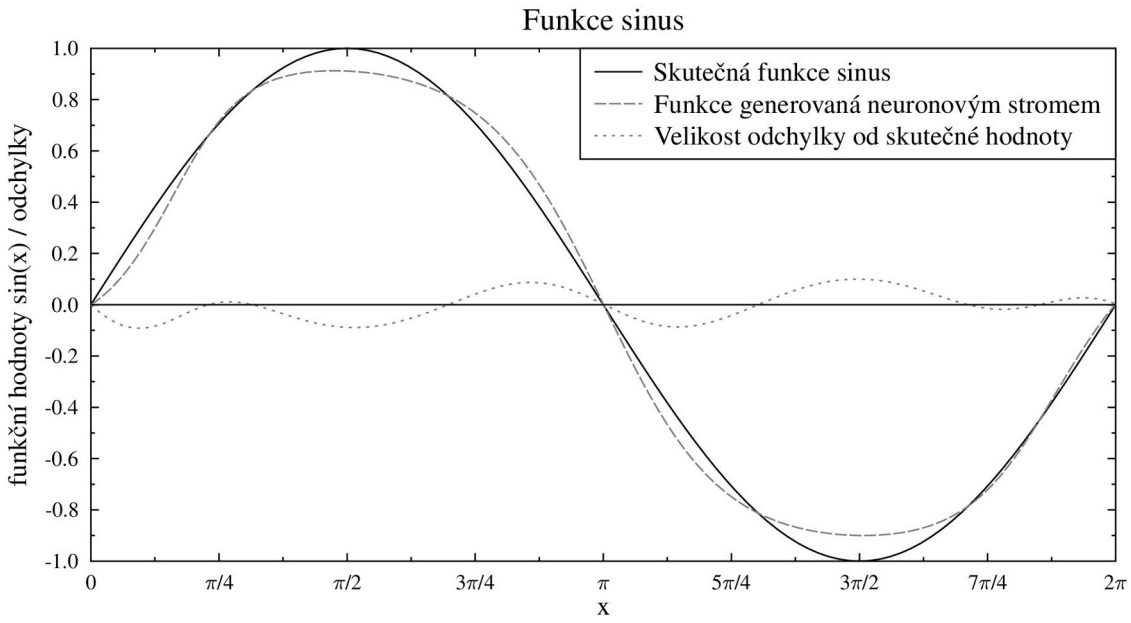
Tabulka 9: Statistika odpovědí pro trénovací množinu - test č. 3

Nejmenší odchylka	0,00271
Průměrná odchylka	0,01433
Median odchylky	0,00798
Maximální odchylka	0,03617

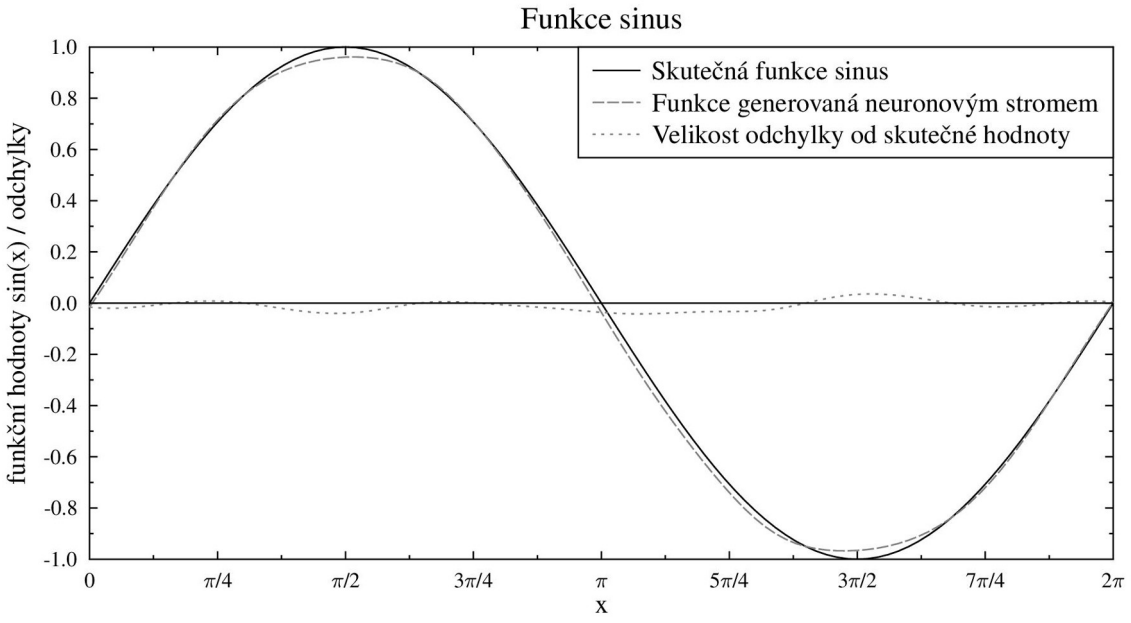
Tabulka 10: Statistika odpovědí pro testovací množinu - test č. 3

V testu č.3 je vidět podstatné zlepšení ve výsledném neuronovém stromu oproti předchozím dvěma neuronovým stromům.

Na závěr použijeme metodu *generateErrorFile()*, která postupně vygeneruje vstupní hodnoty přes celý interval $\langle 0°; 360° \rangle$ v určitých krocích a vypočítá rozdíly v odpovědích neuronového stromu oproti skutečným hodnotám. Tyto rozdíly (chyby) zapíše do souboru CSV. Tento soubor se pak použije pro vytvoření grafu, který nám ukáže odchylky přes celý rozsah. Pro toto porovnání je zvoleno 100 kroků. Na obrázcích 17 - 19 jsou vidět výsledky. Na vodorovných osách jsou vstupní hodnoty (x). Na svislých osách jsou funkční hodnoty a odchylky (chyby) v odpovědích neuronových stromů oproti skutečným hodnotám funkce sinus. Neuronový strom z testu č.1 má v oblasti vstupních hodnot $(\pi/2)$,



Obrázek 18: Odpovědi neuronového stromu z testu č. 2



Obrázek 19: Odpovědi neuronového stromu z testu č. 3

$(7\pi/8)$, $(9\pi/8)$ a $(3\pi/2)$ velké nepřesnosti. Neuronový strom z testu č.2 je na tom lépe, díky jinak zvolené fukci zdatnosti. Neuronový strom z testu č.3 má podstatně přesnější odpovědi na celém intervalu. Tyto tři testy ukazují mimo jiné také vliv správného zvolení trénovací množiny na výsledné chování neuronového stromu.

5.2 Realizace funkce 6x - 3y + z na omezeném intervalu

Problém, kterým je funkce $6x - 3y + z$, se mírně liší od předchozího, kterým byla funkce sinus. Tentokráte máme tři vstupy, kterými jsou proměnné x, y a z. Všechny tři proměnné mohou nabývat libovolných reálných hodnot. My si pro jednoduchost zvolíme testovací interval všech vstupních hodnot na $\langle 0; 1\rangle$. Pouze výstupní proměnná bude v jiném intervalu. Když spočítáme minimum a maximum této funkce na omezeném vstupním intervalu $\langle 0; 1\rangle$, zjistíme, že obor hodnot padne do intervalu $\langle -0,3; 1,6\rangle$. Tento interval tedy budeme muset převést na interval $\langle 0; 1\rangle$, který nám zároveň umožní použít kteroukoli aktivační funkci neuronu z knihovny. Tentokráte tedy budeme konvertovat výstupní hodnoty.

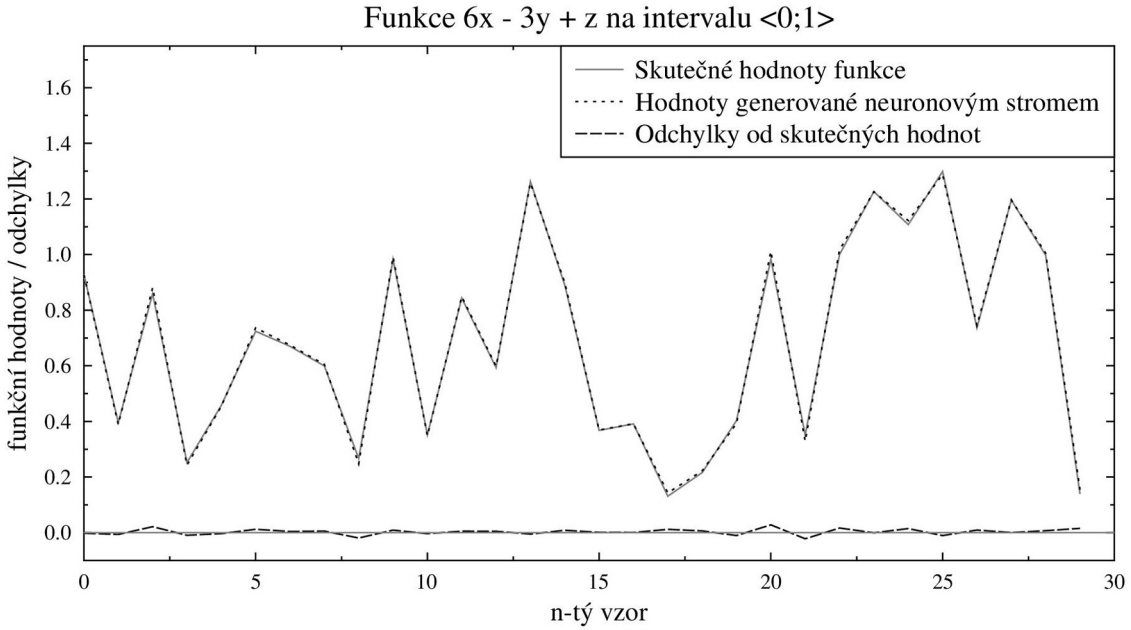
5.2.1 Příprava testovací třídy

Pro tento test byl opět použit samostatný projekt s názvem *Test of FNT library*, ve kterém je pouze jediná třída. Ta obsahuje většinu metod stejných jako v předchozím testu funkce sinus a několik metod mírně odlišných, ale princip je stejný. Trénovací a testovací množinu nebudeme vytvářet manuálně, ale vygenerujeme je pomoci metody *generatePatterns(bool training)*. Hodnotou argumentu *true* řekneme metodě, že požadujeme trénovací množinu, a metoda vygeneruje předem daný počet vzorů, které uloží do souboru CSV pro pozdější použití. Obdobně vygenerujeme i testovací množinu. Trénovací množina je uložena v souboru „training_patterns_A.csv“ a testovací množina v souboru „testing_patterns_A.csv“. Obě množiny obsahují 30 vzorů.

5.2.2 Test č.4

Maximální povolená chyba byla tentokráte nastavena na hodnotu 0,01. Velikost svazku je 15, což při 30 vzorech v trénovací množině znamená 2 svazky + 1 navíc, který si knihovna přidává sama. Multiplikátor je taktéž 2 a počet vstupů 3. Při tomto nastavení máme maximální počet neuronů ve skrytých vrstvách 18. Celkově tedy může neuronový strom obsahovat 22 neuronů. Aktivační funkce byla změněna na *Radiální bázi*. Také velikost populace byla snížena na 100.

Výsledky procesu
Proces běžel 307 sekund a vygeneroval 21 506 neuronových stromů v 16 epochách. Nejlepší z nich se skládá z 12 neuronů a je uložen v souboru *xyz_function_tree_1.fnt*. Z tabulek 12 a 13 jsou opět vidět statistiky procesu, které říkají, že tentokrát je maximální chyba v odpovědi neuronového stromu pro testovací množinu téměř 1,5x vyšší než povolená. To je rozdíl od předchozích testů funkce sinus, kde neuronový strom i pro testovací



Obrázek 20: Výsledky neuronového stromu z testu č. 4 pro testovací množinu.

MaxAllowedError	0,01
SizeOfBunch	15
Multiplicator	2
LearningRate	0,2
PreviousDeltaRate	0,5
MaxCyclesCount	200
Function	RADIAL_BASIS_FUNCTION
Fitness	MAX_ERROR
PopulationSize	100
MaxGenerations	100
MaxEpochs	100

Tabulka 11: Nastavení parametrů třídy *Task* - test č. 4

Nejmenší odchylka	1,778 E-05
Průměrná odchylka	0,00286
Median odchylky	0,00212
Maximální odchylka	0,00999

Tabulka 12: Statistika odpovědí pro trénovací množinu - test č. 4

Nejmenší odchylka	0,00015
Průměrná odchylka	0,00481
Median odchylky	0,00446
Maximální odchylka	0,01476

Tabulka 13: Statistika odpovědí pro testovací množinu - test č. 4

množinu dával výsledky s podobnou přesností, jako pro trénovací množinu. Důvodem je

trénovací množina. V případě funkce sinus byla trénovací množina vytvořena manuálně s úmyslem rovnoměrně pokrýt celý vstupní interval. Kdežto v tomto testu pro funkci $6x - 3y + z$ byla trénovací množina vygenerována náhodně, takže není zajištěno pokrytí celého vstupního intervalu rovnoměrně. Tady je vidět, jak příprava trénovací množiny může ovlivnit výsledný neuronový strom.

5.2.3 Test č.5

MaxAllowedError	0,001
SizeOfBunch	5
Multiplicator	1
LearningRate	0,15
PreviousDeltaRate	0,3
MaxCyclesCount	200
Function	RADIAL_BASIS_FUNCTION
Fitness	MAX_ERROR
PopulationSize	500
EliteSize	50
MaxGenerations	100
MaxEpochs	500

Tabulka 14: Konečné nastavení parametrů třídy *Task* - test č. 5

V tomto testu se pokusíme dosáhnout větší přesnosti v odpovědích neuronového stromu. Maximální povolenou chybu tedy nastavíme z hodnoty 0,01 na hodnotu 0,001, tj. o jeden řád nižší.

Výsledky procesu
Proces bežel 2673 sekund (asi 45 minut), prošel všech 100 epoch, ale řešení v požadované toleranci chyby nenašel. Nejvhodnější neuronový strom měl chybu 0,002248. Byl tedy zvýšen maximální počet epoch na hodnotu 500. Po 250 epochách bylo vidět, že je chyba nižší (0,001797), ale rychlost, s jakou se snižovala byla malá. Pravděpodobně se

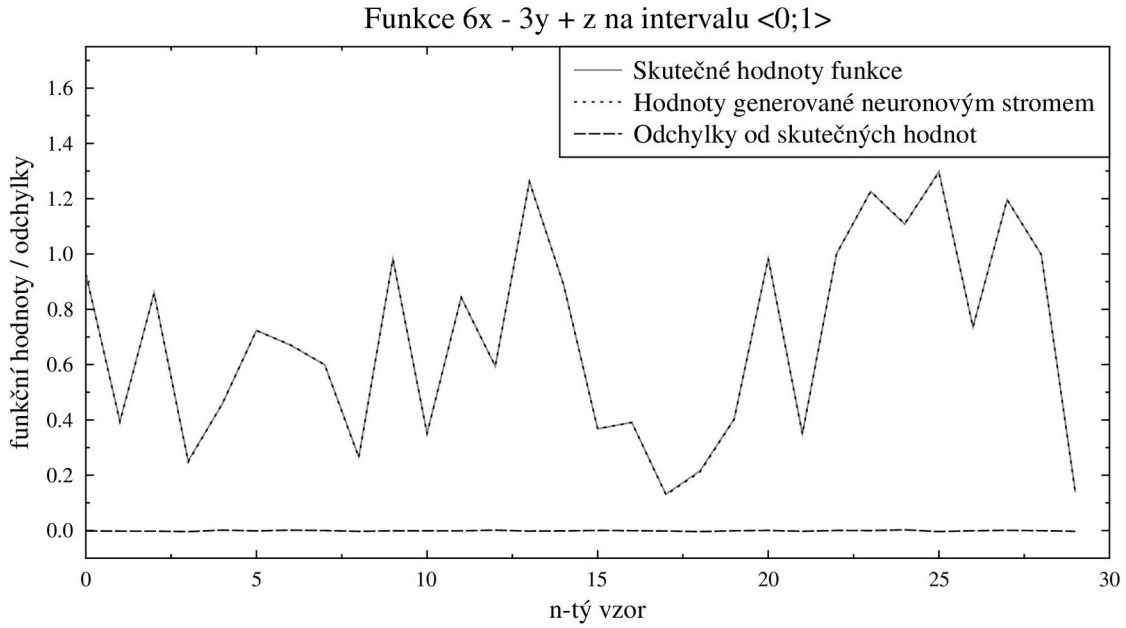
Nejmenší odchylka	6,653 E-06
Průměrná odchylka	0,00055
Median odchylky	0,00063
Maximální odchylka	0,00099

Tabulka 15: Statistika odpovědí pro trénovací množinu - test č. 5

Nejmenší odchylka	5,734 E-05
Průměrná odchylka	0,00083
Median odchylky	0,00073
Maximální odchylka	0,00214

Tabulka 16: Statistika odpovědí pro testovací množinu - test č. 5

proces zabývá zbytečně dlouhou dobu učením nevhodných struktur. Také je možné, že výchozí počet elitních jedinců 6 je málo. Proto byla zvýšena velikost populace na hodnotu 500, počet elitních jedinců na hodnotu 50 a proces spuštěn znovu s úmyslem jej urychlit.



Obrázek 21: Výsledky neuronového stromu z testu č. 5 pro testovací množinu.

Tímto se zvýší počet jedinců testovaných v jedné epoše, což by mohlo pomoci, protože výpočetní nároky genetického algoritmu jsou nižší než výpočetní nároky učení.

Změna byla pozorována už na začátku procesu. Chyba nejlepšího jedince klesala podstatně rychleji. To se však později změnilo a proto byl proces přerušen. Po několika experimentech s nastavováním parametrů bylo dosaženo chyby 0,00099. Proces trval 6 hodin a vygeneroval 2 102 494 různých struktur ve 330 epochách. Zvyšování maximálního počtu neuronů ve stromu jen pomoci násobitele nebylo zrovna vhodné, jelikož se do procesu dostávaly i struktury s malým počtem neuronů (typicky 6). Tyto struktury na začátku „rychle“ konvergovaly k hodnotám chyby okolo 0,002, ale pak už se nebyly schopny dostat na nižší hodnotu. Násobitel byl tedy nastaven na hodnotu 1, čímž se ve výpočtu nijak neprojevil a velikost svazku byla snížena na hodnotu 5. Takto bylo dosaženo hodnoty maximálního počtu neuronů ve skrytých vrstvách 21, což se moc neliší od původního počtu 18 v předchozím testu. Zároveň se však zvýšil minimální počet neuronů na hodnotu 7. Další problém, který se objevil, způsoboval „rozkmitávání“ v nastavení vah při učení. Byl proto snížen koeficient učení na hodnotu 0,15 a vliv předchozího kroku na hodnotu 0,3. Výsledné nastavení parametrů třídy *Task* ukazuje tabulka 14. Neuronový strom se kládá z 12 neuronů a je uložen v souboru *xyz_function_tree_2.fnt*. Tabulky 15 a 16 ukazují statistiky výsledků. Opět je vidět, že pro některé vzory testovací množiny jsou odpovědi mírně horší. Obrázky 20 a 21 zobrazují průběh funkce na intervalu $\langle 0; 1 \rangle$ a chyby v odpovědích neuronových stromů z testů č. 4 a 5 pro testovací množinu.

5.3 Řízení softwarového auta

Předchozí testy se zabývaly oblastí zvanou aproximace funkce. Na dvou matematických funkcích bylo předvedeno základní použití neuronových stromů a také jak použít implementovanou knihovnu k jejich nalezení či vytvoření. Tento test předvede jinou oblast využití neuronového stromu, kterou je řízení auta. Nejedná se o fyzické auto, ale o jeho softwarovou emulaci. K tomuto testu využijeme systém pro závodění aut řízených umělou inteligencí, který vyvinul Marian Krucina v rámci své diplomové práce na VŠB-TUO [6]. Systém pro závodění aut je typu klient - server. Server je v tomto případě systém samotný, klient je pak jakákoli aplikace, která se k systému umí připojit a komunikovat s ním. Závod pak probíhá tak, že aplikace přijímá v cyklu informace o stavu auta, tyto informace vyhodnocuje a odesílá na ně odpověď. Systém byl vyvinut za účelem testování umělých inteligencí, takže je vhodý i pro otestováni FNT a zároveň tak i implementované knihovny.

5.3.1 Popis komunikace při závodu

Systém pošle klientovi informaci o aktuální pozici auta, klient tuto informaci vyhodnotí a pošle zpět svou odpověď jako reakci. Podle reakce klienta pak systém provede určité činnosti, simulující řízení auta, které dostanou auto do jiné pozice. O nové pozici auta systém opět informuje klienta. Komunikace tak probíhá v neustálem cyklu až do ukončení závodu. Informace o pozici auta na cestě se skládá ze čtyř částí:

- aktuální vzdálenost auta od středové čáry na cestě
- aktuální úhel, který svírá osa auta a osa středové čáry cesty (směr auta)
- aktuální rychlost auta
- budoucí vzdálenost auta od středové čáry na cestě za 1 sekundu, nezmění-li po tuto dobu rychlost a směr

Reakce klienta obsahuje pouze dvě částí informace:

- úroveň sešlápnutí plynového nebo brzdového pedálu
- úroveň natočení kol

Tento problém tedy představuje zobrazení ze čtyřdimenzionálního prostoru do dvoudimenzionálního. Všechny vyjmenované parametry, které představují informace mezi serverem a klientem jsou zakódované do reálných čísel v intervalu $\langle 0; 1 \rangle$. Více informací lze nalézt v již zmiňované diplomové práci [6].

V tomto případě neznáme dopředu funkci, kterou má neuronový strom realizovat, tak jako tomu bylo v předchozích dvou případech. Vzory trénovací množiny tak musíme připravit na základě zkušeností s řízením auta. Je potřeba analyzovat chování systému pro závodění aut a vymyslet vhodné trénovací vzory. Je jasné, že nemůžeme pokrýt trénovacími vzory rovnoměrně celý vstupní interval, neboť by těchto vzorů bylo skutečně

mnoho. Navíc ani není v silách člověka rozpoznat všechny situace, do kterých se auto v závodu může dostat. Je potřeba tedy dobře zvážit, jak vybrat jednotlivé trénovací vzory tak, aby dostatečně reprezentovaly schopnost řídit auto.

5.3.2 Příprava testovacího projektu

Tentokrát nebude stačit pouze jedna třída. Je potřeba vyvořit aplikaci, která se připojí k závodu na serveru, vytvoří závod a postaví auto do závodu. Projekt pro tyto účely je pojmenován *NeuroRaces*. Je to GUI aplikace, která umožňuje vytvořit dvojci neuronových stromů se čtyřmi vstupy, tuto dvojici přihlásit do závodu, v reálném čase komunikovat se serverem a reagovat na aktuální situaci v závodu. Testování pak spočívá v tom, že po nalezení vhodného řešení ve formě dvojice neuronových stromů, se nechají tyto stromy odpovídat (reagovat) na situaci v konkrétním závodu a podle dokončení nebo nedokončení závodu ověřit jejich schopnosti. Tentokrát totiž nemáme možnost porovnat chování neuronového stromu s předem vypočítanými výsledky. Musíme tedy výsledek procesu hledání vhodného řešení ověřit manuálně a to pozorováním závodu a analýzou chyb, kterých se auto při jízdě dopouští.

5.3.3 Test č. 6

Vzdálenost	Úhel	Rychlost	Vzdálenost 2	Plyn/Brzda	Kola
0.50	0.50	0.50	0.50	1.00	0.50
0.52	0.51	0.70	0.52	1.00	0.48
0.48	0.49	0.70	0.48	1.00	0.52
0.45	0.45	0.60	0.40	1.00	0.55
0.55	0.55	0.60	0.60	1.00	0.45
0.40	0.40	0.60	0.35	0.95	0.60
0.60	0.60	0.60	0.65	0.95	0.40
0.40	0.40	0.90	0.3	0.92	0.65
0.60	0.60	0.90	0.70	0.92	0.35

Tabulka 17: Trénovací množina - test č. 6

Tabulka 17 představuje trénovací množinu. První sloupec představuje aktuální vzdálenost auta od středové čáry. Hodnota 0,5 znamená že je auto na čáře. Nižší hodnota ukazuje jak moc je auto vlevo, vyšší hodnota pak jak moc je auto vpravo. Druhý sloupec představuje úhel, který svírá osa auta s osou středové čáry. Hodnota 0,5 znamená že obě osy jsou rvnoběžné (auto jede rovně). Hodnota nižší ukazuje o kolik je osa auta pootočená vlevo (auto jede na levou stranu). Vyšší hodnota pak znamená že auto jede na pravou stranu cesty. Třetí sloupec představuje aktuální rychlost auta. Hodnota 0,5 ukazuje, že auto stojí, hodnota nižší, s jakou rychlostí couvá, hodnota vyšší, jak rychle jede dopředu. Čtvrtý sloupec představuje vzdálenost auta od středové čáry za jednu sekundu (nezmění-li auto rychlost a směr). Hodnota 0,5 znamená, že auto bude na středové čáře, hodnota nižší jak moc bude auto vlevo od čáry, hodnota vyšší, jak moc bude vpravo od čáry. Pátý

sloupec představuje odpověď, která říká, jestli se má sešlápnout plyn nebo se má brzdit. Hodnota 0,5 znamená že se neděje nic. Hodnota nižší představuje intenzitu brždění, hodnota vyšší představuje polohu sešlápnutí plynového pedálu. Poslední sloupec pak představuje natočení kol. Hodnota 0,5 znamená, že mají kola zůstat v přímém směru, hodnota nižší znamená, o jaký úhel se mají natočit vlevo, hodnota vyšší pak, o jaký úhel vpravo. Levá část prvního řádku tabulky tedy říká, že auto stojí na středové čáře s kolama natočenýma v přímém směru (informace o stavu). Pravá část prvního řádku tabulky pak říká, že kola mají zůstat natočené stejně a má se sešlápnout plyn na maximum (odpověď na současný stav).

MaxAllowedError	0,03
SizeOfBunch	10
Multiplicator	2
LearningRate	0,2
PreviousDeltaRate	0,5
MaxCyclesCount	200
Function	RADIAL_BASIS_FUNCTION
Fitness	MAX_ERROR
PopulationSize	500
EliteSize	50
MaxGenerations	200
MaxEpochs	500

Tabulka 18: Nastavení parametrů třídy *Task* - test č. 6

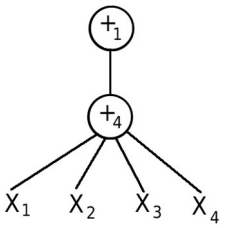
V tabulce 18 je nastavení úkolu pro vytvoření dvou neuronových stromů představujících řidiče. Velikost maximální povolené chyby je potřeba experimentálně zjistit. Příliš velká chyba způsobí neschopnost řízení. Příliš malá chyba pak „podivné“ chování v situaci, na kterou nebyl řidič učen. Jelikož trénovací množina v tomto případě obsahuje velice málo trénovacích vzorů vzhledem k velikosti vstupního prostoru, je lepší umožnit větší chybu v odpovědích neuronové sítě. Hyperplocha tvořená jednotlivými váhami a prahy neuronů se tak „rozprostře“ mezi body v prostoru (trénovací vzory) rovnoměrněji. Vpřípadě trvání na příliš malé chybě se hyperplocha bude snažit v místech bodů trénovacích vzorů těmto bodům více přiblížit a výsledkem pak může být její větší zvlnění. To se projeví zhoršenou schopností zobecňování neboli přeučením, kdy neuronová síť bude schopna přesně odpovědět na naučené vzory, ale na nenaučené vzory už ne.

Po dokončení procesu je možné kromě řidiče vytvořit i soubor s informacemi o tom, jak proces skončil. V tomto testu toho využijeme.

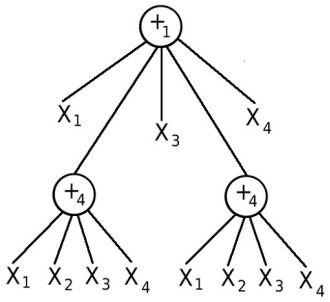
Výsledky procesu
Tentokrát je proces složen ze dvou podprocesů, protože hledáme dva neuronové stromy. Jeden pro určování natočení kol a druhý pro řízení rychlosti. Neuronový strom pro řízení rychlosti zabral 1,1 sekundy a bylo při tom vygenerováno 2 236 neuronových stromů.

Bylo dosaženo maximální chyby 0,0158 při čtyřech použitých neuronech. Zde je vidět, že neuronový strom si ze čtyř vstupů vybral pouze dva. Vzdálenost a vzdálenost 2. Druhý strom pro řízení kol zabral 17 sekund a bylo při tom vygenerováno 14 434 stromů. Zde bylo dosaženo maximální chyby 0,0222 při pěti neuronech. Tentokrát se nepoužil vstup představující rychlost. Neuronové stromy tohoto řešení jsou uloženy v souboru *Test driver radial 1.fnt*. Informace o průběhu hledání struktury pak v souboru *Test driver radial 1_debug.fnt*.

5.3.4 Test č. 7



Obrázek 22: Neuronový strom pro řízení rychlosti



Obrázek 23: Neuronový strom pro řízení kol

V tomto testu pouze snížíme maximální chybu na hodnotu 0,01 a necháme vytvořit druhého řidiče. Pak oba výsledky (řidiče) porovnáme mezi sebou.

Výsledky procesu
Vytvoření stromu pro řízení rychlosti trvalo 17 sekund. Prošlo se jednou epochou a bylo vytvořeno 13 096 stromů. Strom je tvořen šesti neurony. Druhý strom trval 102 sekund, prošly se 3 epochy a vygenerovalo 52 030 stromů. Bylo potřeba sedm neuronů. Neuronové stromy tohoto řešení jsou uloženy v souboru *Test driver radial 2.fnt*. Informace o průběhu hledání struktury pak v souboru *Test driver radial 2_debug.fnt*.

Porovnání řidičů z předchozích testů První řidič (test č. 6) je nepatrně rychlejší v zatáčkách, ale nedrží se tolik středové čáry. Druhý řidič je pomalejší, ale přesněji projíždí úseky a je si na trati jistější (nekmitá tolik kolami). Oba řidiči jsou však schopni jízdy.

6 Závěr

Cílem práce bylo naimplementovat knihovnu pro práci s neuronovou sítí typu Flexibilní neuronový strom (FNT) v prostředí .NET. Tento cíl byl splněn. Knihovna umožňuje připravit zadání úkolu (popis problému), dle tohoto zadání vytvořit jeden nebo více neuronových stromů a pak s tímto neuronovým stromem či více stromy pracovat. Zadání obsahuje několik parametrů, pomocí kterých lze ovlivnit průběh zpracování úkolu (hledání vhodného neuronového stromu či stromů). Nalezení vhodné struktury neuronového stromu bylo vyřešeno použitím genetického programování. Pro optimalizaci parametrů je pak použito parametrické verze metody zpětného šíření. Dále je knihovna navržená tak, aby mohla být začleněna do externí aplikace a s touto aplikací jednoduchým způsobem komunikovat. To umožňuje práci s knihovnou automatizovat. Model neuronového stromu je oddělen od optimalizačních algoritmů, což umožňuje vytvořené neuronové stromy snadno ukládat do souboru XML a zpětně je načítat. Kromě knihovny samotné byly vytvořeny tři malé aplikace, které slouží k jejímu základnímu otestování. Tyto aplikace jsou obsahem práce, včetně souborů, které byly použity při testování knihovny.

Testy knihovny a neuronových stromů touto knihovnou vytvořených ukazují, že knihovna funguje. Dále také ukazují důležitost přípravy trénovací množiny, která se odráží na celkovém výsledku vytvářeného neuronového stromu. Nevhodně zvolená trénovací množina má nepříznivý dopad na chování výsledného neuronového stromu, což může vzbuzovat dojem, že knihovna nedává očekávané výsledky. Testy tak potvrzují některé části teorie popsané v kapitole 2, například problém vhodné volby trénovcích vzorů. Při testování knihovny jsem experimentoval s nastavováním všech parametrů na různé hodnoty a vypozoroval jisté závislosti v chování procesu. Tyto poznatky mě vedly k přednastavení každého parametru na určitou výchozí hodnotu.

V průběhu práce jsem narazil na jisté problémy, jejichž vyřešení by ulehčilo práci s knihovnou. Jedním z nich je přibližný odhad počtu potřebných neuronů pro konkrétní zadání. Momentálně je tohle řešeno částečně na základě informací od uživatele knihovny. Možná by se to dalo řešit analýzou rozložení dat v prostoru. Data v prostoru zde představují trénovací vzory. Uživatel by nemusel přemýšlet nad počtem potřebných neuronů a proces by se mohl urychlit. Další vylepšení knihovny může spočívat v úpravě funkce zdatnosti. Nyní se vyhodnocuje jako chyba odezvy. Kdyby se přidalo ještě porovnávání velikosti stromů podle počtu neuronů, dosáhlo by se tím upřednostňování menších struktur.

Z publikací, které jsem v průběhu této práce pročetl, vyplývá, že flexibilní neuronové stromy mají velký potenciál v mnoha oblastech. Nové studie ukazují jejich použití nejen v oblastech financí ale i ve zdravotnictví.

7 Reference

[1] Abraham, Ajith a Chen Yuehui: *Tree-Structure Based Hybrid Computational Intelligence*. 2, 2009. http://link.springer.com/book/10.1007/978-3-642-04739-8/page/1.

[2] Biskup, Roman: *Možnosti neuronových sítí*. 2009.

[3] Bouaziz, Souhir, Alimi Adel M a Abraham Ajith: *Evolving Flexible Beta Basis Function Neural Tree for Nonlinear Systems*. 2013.

[4] Chen, Yuehui, Yang Bin a Meng Qingfang: *Small-time scale network traffic prediction based on flexible neural tree*. Applied Soft Computing, 12(1):274 – 279, 2012, ISSN 1568-4946. http://www.sciencedirect.com/science/article/pii/S1568494611003280.

[5] Görlichová, Lucie: *Umělé neuronové sítě v lékařské diagnostice*. diplomová práce, Masarykova univerzita v Brně, Přírodovědecká fakulta, 2006. http://is.muni.cz/th/52088/prif_m/Text_prace.pdf.

[6] Krucina, Marian: *Závody aut řízených umělou inteligencí*. diplomová práce, VŠB - Technická univerzita Ostrava, 2008. http://hdl.handle.net/10084/66742.

[7] Kuba, Martin: *Neuronové sítě*. 1995.

[8] Šíma, Jiří a Neruda Roman: *Teoretické otázky neuronových sítí*. 1996.

[9] Peng, Lizhi, Yang Bo, Zhang Lei a Chen Yuehui: *A parallel evolving algorithm for flexible neural tree*. 37:653–666, 2011. http://www.sciencedirect.com/science/article/pii/S0167819111000615.

[10] Skopal, Tomáš: *Neuronové sítě a Information Retrieval*. www.cs.vsb.cz/arg/techreports/neural.pdf.

[11] Vojáček, Antonín: *Samoučící se neuronová síť - SOM, Kohonenovy mapy*. 2006. www.kiv.zcu.cz/studies/predmety/uir/NS/Samouc_NN2.pdf.

[12] Volná, Eva: *Neuronové sítě 1*. 2002.

[13] Vondrak, Ivo: *Umělá inteligence a neuronové sítě*, 2001.

[14] Wang, QingHua, YiNa Guo a Ajith Abraham: *Online Hand Gesture Recognition Using Surface Electromyography Based on Flexible Neural Trees*. V Deng, Hepu, Duoqian Miao, Jingsheng Lei a FuLee Wang (editoři): *Artificial Intelligence and Computational Intelligence*, svazek 7004 z *Lecture Notes in Computer Science*, strany 245–253. Springer Berlin Heidelberg, 2011, ISBN 978-3-642-23895-6. http://dx.doi.org/10.1007/978-3-642-23896-3_29.

[15] Wang, Yu: *Study on Improved Flexible Neural tree Optimization Algorithm*. 2013. http://www.atlantis-press.com/php/download_paper.php?id=5910.

[16] Zhang, Byoung tak, Ohm Peter a Mühlenbein Heinz: *Evolutionary induction of sparse neural trees*. Evolutionary Computation, 1997.

A CD s elektronickou verzí diplomové práce

A.1 Implementace knihovny FNT

Soubor s názvem *FlexibleNeuralTree.dll*.

A.2 Zdrojové kódy knihovny FNT

Kompletní projekt pro Microsoft Visual Studio 2010 - adresář *Zdrojové kódy knihovny FNT*.

A.3 Zdrojové kódy testovacích aplikací

Kompletní projekty pro Microsoft Visual Studio 2010 - adresář *Zdrojové kódy testovacích aplikací*.

- *NeuroRaces*
- *Test of FNT library*
- *TestFntLibrarySinus*

A.4 XML soubory s uloženými neuronovými stromy

Adresář s názvem *Soubory neuronových stromů*. Přiloženy jsou i CSV soubory s vygenerovanými hodnotami pro vytvoření grafů.

A.5 Použité obrázky

Adresář *Použité obrázky*.

A.6 Uživatelská příručka FNT

Soubor s názvem *FNT uživatelská příručka.pdf*.